Betriebssysteme (BS)

VL 1 – Einführung

Volkmar Sieh / **Daniel Lohmann**

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 18. Oktober 2023

`https://sys.cs.fau.de/lehre/ws23/bs`

Vorwort

Die Lehrveranstaltung ist grundsätzlich für alle Studiengänge offen. Sie verlangt allerdings gewisse Vorkenntnisse. Diese müssen nicht durch Teilnahme an den Lehrveranstaltungen von I4/I16 erworben worden sein.

Lernziele

- **Vertiefen** des Wissens über die interne Funktionsweise von Betriebssystemen
  - Ausgangspunkt: Systemprogrammierung
  - Schwerpunkt: Nebenläufigkeit und Synchronisation
- **Entwickeln** eines Betriebssystems *von der Pike auf*
  - OOStuBS / MPStuBS Lehrbetriebssysteme
  - Praktische Erfahrungen im Betriebsystembau machen
- **Verstehen** der technologischen Hardware-Grundlagen
  - PC-Technologie verstehen und einschätzen können
  - Schwerpunkt: x86-/x86_64-Architektur

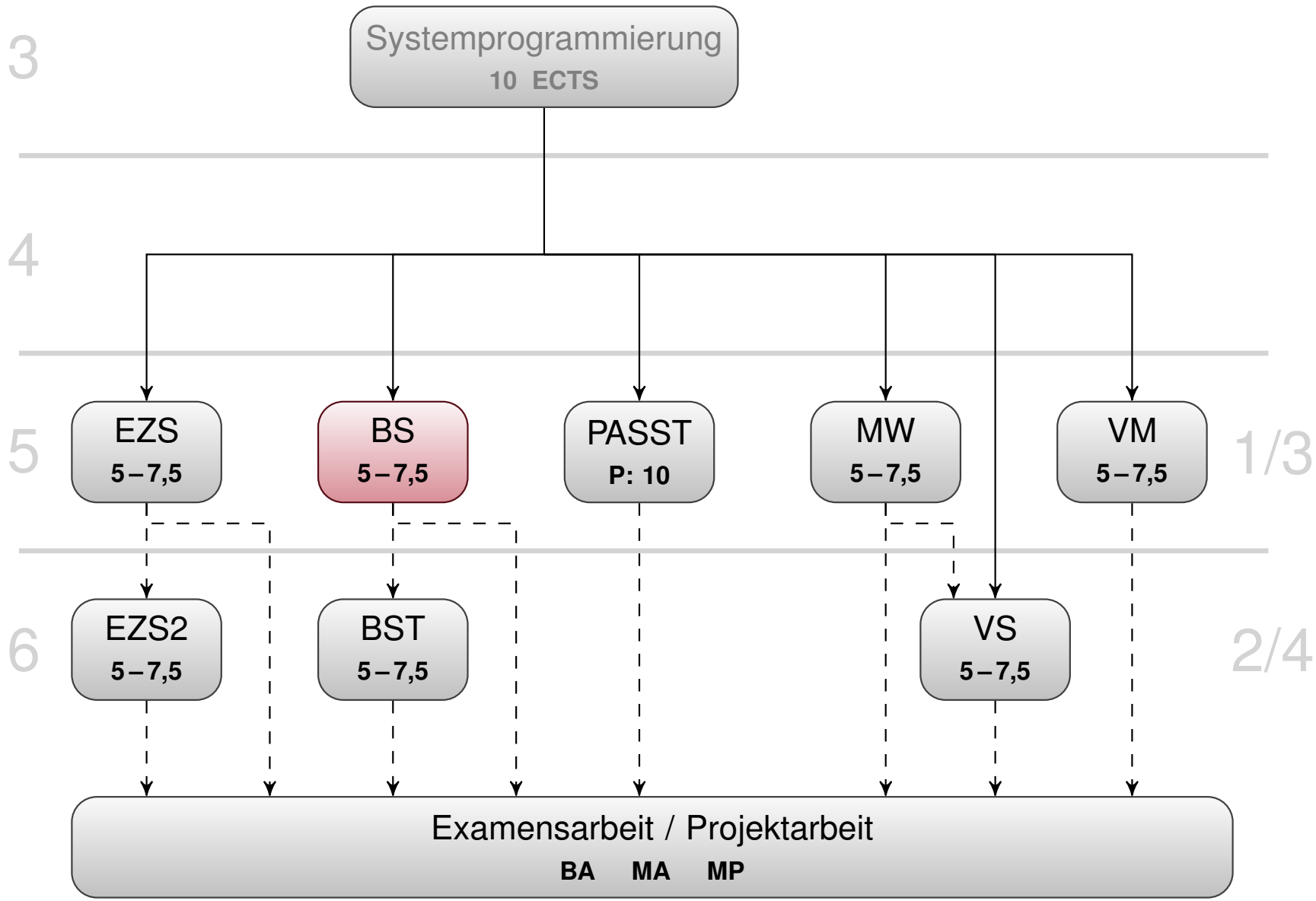
Voraussetzungen

- Rechnerarchitektur, **Systemprogrammierung**
- C / **C++**, Assembler (x86/x86_64)
- Ein gewisses Maß an **Durchhaltevermögen**
- Freude an systemnaher und **hardwarenaher Programmierung**

Wir arbeiten auf der “nackten Maschine” (*bare metal*)!

Die meisten sind überrascht, wie viel Spaß das macht :-)

Einpassung in den Musterstudienplan (Bachelor/Master)



Veranstaltungsbestandteile

(5 – 7,5 ECTS Punkte)

VL – Vorlesung **2,5**

Vorstellung und detailierte Behandlung des Lehrstoffs

+

Ü – Übung **2,5**

- Übung **OOStuBS**
- 6 Übungsaufgaben
- Abnahme alle 14 Tage

oder

EÜ – Erweiterte Übung **5**

- Übung **MPStuBS**
- 6 erweiterte Aufgaben
- Abnahme alle 14 Tage

Verwendbarkeit, Scheinerwerb und Modulnote

- **Wahlpflichtmodul** (Bachelor/Master) der Vertiefungsrichtung **Verteilte Systeme und Betriebssysteme**
  - eigenständig (nur BS) — VL + Ü oder VL + EÜ
  - mit weiteren Veranstaltungen — VL oder VL + Ü oder VL + EÜ
- Studien- und Prüfungsleistungen
  - Bachelor — Prüfungsleistung
  - Master — Prüfungsleistung
    erworben durch
    - erfolgreiche Teilnahme an den Übungen
    - erfolgreiche Bearbeitung aller Übungsaufgaben
    - 30 min. mündliche Prüfung
- Berechnung der Modulnote
  - Note der mündlichen Prüfung + “Übungsbonus” in Zweifelsfällen

Übungsbetrieb

Tafelübung

Raum 0.031 (Aquarium)

- **Ein** Termin
  - Mi, 16:15 – 17:45
- Übungsaufgaben sind in 2er-Gruppen zu bearbeiten

Rechnerübung

Raum 01.153 (WinCIP)

- Zwei Termine
  - Mi, 14:00 – 16:00
  - Fr, 12:00 – 14:00
- Abgabe der Übungsaufgaben

Zusatzseminar (freiwillig)

Raum 0.031 (Aquarium)

- Drei Einzeltermine
  - Mi, 08.11., 22.11. und 06.12. um 16:15 Uhr
  - Zeitschlitz der Tafelübung

Terminübersicht Wintersemester 2023

KW	Mi 12-14	Mi 14-16	Mi 16-18	Fr 12-14	Raum
42	VL_1	RÜ			0.031
43	VL_2	RÜ	$Ü_1$	RÜ	
44				RÜ	0.031
45	VL_3	RÜ	Sem_1	A_1	0.031
46	VL_4	RÜ	$Ü_2$	RÜ	
47	VL_5	RÜ	Sem_2	A_2	01.153
48	VL_6	RÜ	$Ü_3$	RÜ	01.153
49	VL_7	RÜ	Sem_3	A_3	
50	VL_8	RÜ	$Ü_4$	RÜ	
51	VL_9	RÜ		A_4	
02	VL_{10}	RÜ	$Ü_5$	RÜ	
03	VL_{11}	RÜ		A_5	
04	VL_{12}	RÜ	$Ü_6$	RÜ	
05	VL_{13}	RÜ		A_6	
06	VL_{14}		$Ü_7$		

Beteiligte Personen

Dozent Vorlesung

Volkmar Sieh

Übungsleiter

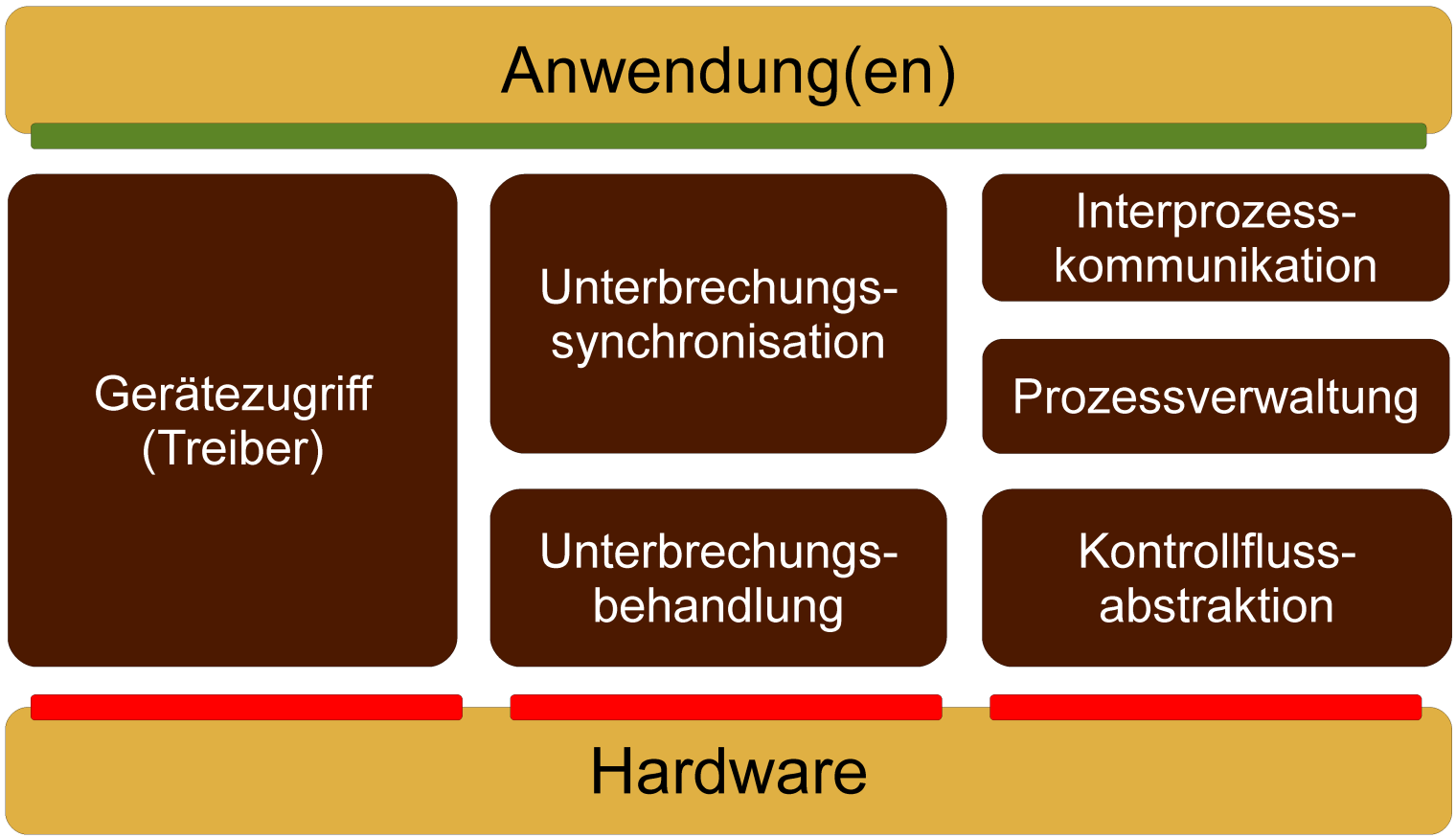
Phillip Raffeck

Dustin Nguyen

Maximilian Ott

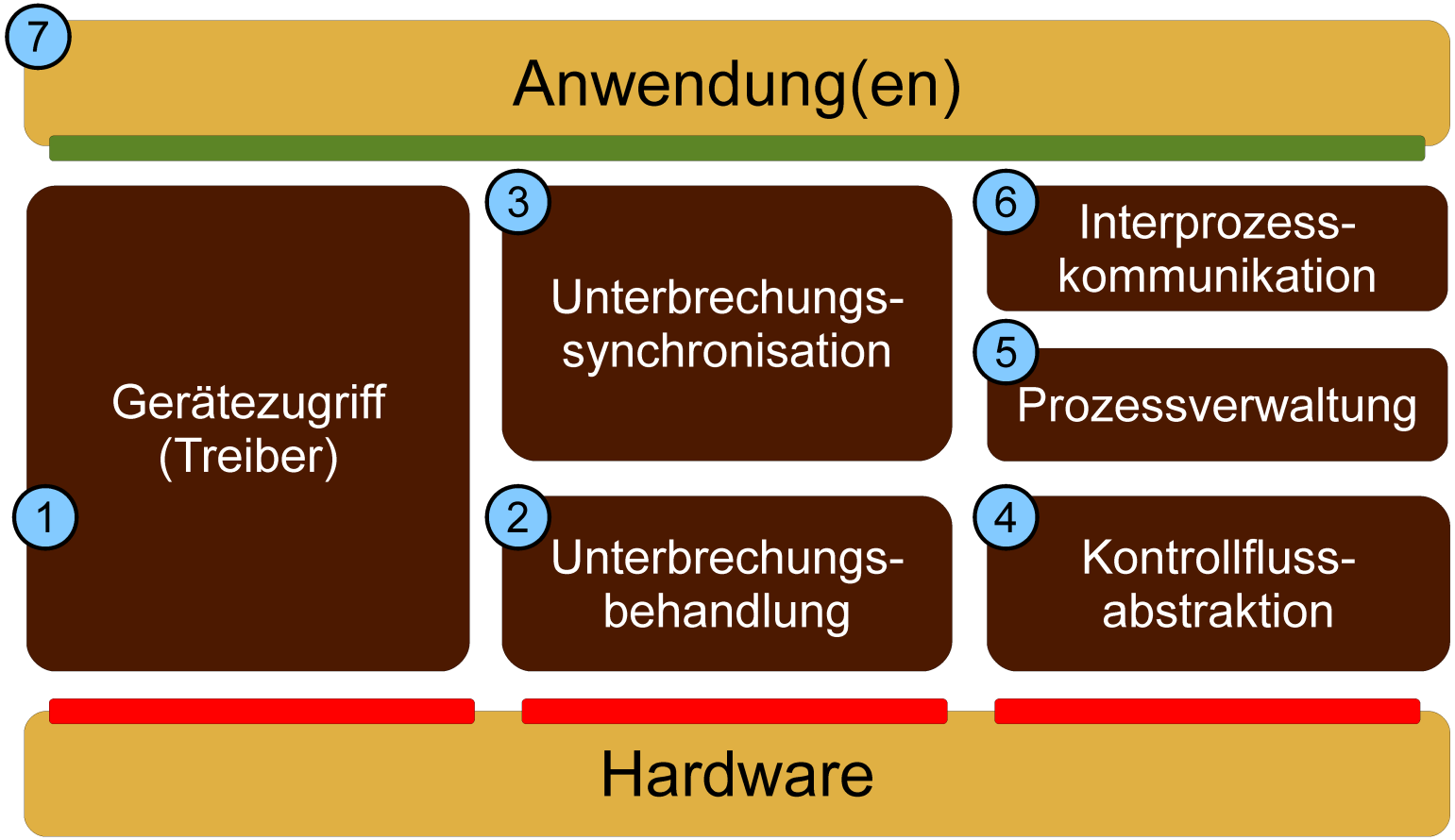
Paul Bergmann

Aufbau eines Betriebssystem



Themenübersicht Übung

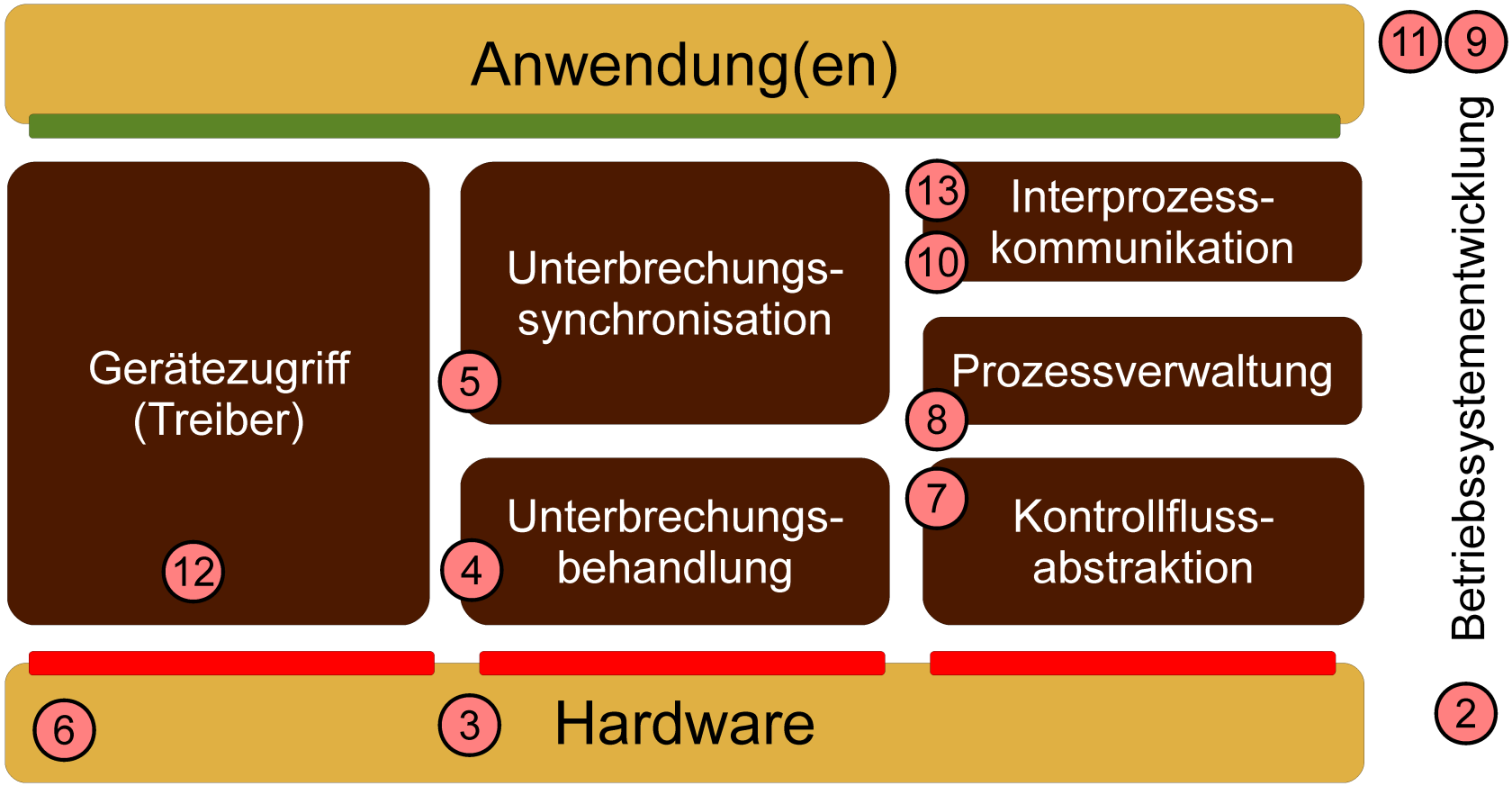
Am Beispiel von: OOStuBS, MPStuBS



Betriebssystementwicklung

Themenübersicht Vorlesung

Am Beispiel von: x86, MC68k, TriCore; Windows, Linux



Verzahnung von Vorlesung und Übungsaufgaben

KW 42	VL_1	*Einführung*	$Ü_0$ *Einführung*
KW 43	VL_2	*BS-Entwicklung*	$Ü_1$ *Ein-/Ausgabe*
KW 44	VL_3	*IRQs (Hardware)*	
KW 45	VL_4	*IRQs (Software)*	
KW 46	VL_5	*IRQs (SoftIRQ)*	$Ü_2$ *IRQ-Behandlung*
KW 47	VL_6	*IRQs (Synchronisation)*	
KW 48	VL_7	*Intel IA-32*	$Ü_3$ *IRQ-Synchronisation*
KW 49	VL_8	*Koroutinen und Fäden*	
KW 50	VL_9	*Scheduling*	$Ü_4$ *Fadenumschaltung*
KW 02	VL_{10}	*Architekturen*	$Ü_5$ *Zeitscheiben-Scheduling*
KW 03	VL_{11}	*Fadensynchronisation*	
KW 04	VL_{12}	*Gerätetreiber*	$Ü_6$ *Fadensynchronisation*
KW 05	VL_{13}	*IPC*	
KW 06	VL_{14}	*Ausblick*	$Ü_7$ *“Eine Anwendung” (opt.)*

VL_2 Betriebssystementwicklung (oft ein harter Kampf!)

- Erste Schritte

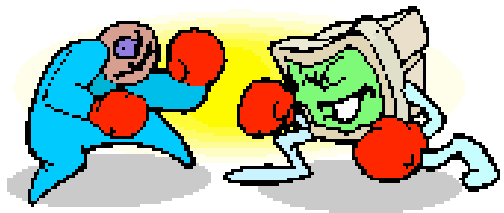
  Wie bringt man sein System auf die Zielhardware?

  - Übersetzen und Linken für “nackte Hardware”
  - Bootvorgang

- Testen und Fehlersuche

  Was tun, wenn das System nicht reagiert?

  - “printf”-*Debugging*
  - Simulatoren
  - *Debugger*
  - *Remote debugging*
  - Hardwareunterstützung



VL_{3-5} Programmunterbrechungen

- im Prinzip
  - Unterbrechungen, *Traps* und Ausnahmen
  - Vektortabellen
  - geschachtelte Unterbrechungen
  - *spurious interrupts*
- beim PC
  - CPU und APIC
  - Unterbrechungen in Multiprozessorsystemen
- Behandlung im Betriebssystem
  - Kopplungsfunktion
  - Zustandssicherung

VL_6 Unterbrechungssynchronisation

- Zusammenspiel zwischen Unterbrechungsbehandlung und “normalem” Kontrollfluss
  - Ursache und Problem
  - Kontrollflussebenenmodell
- Hardware-Mechanismen zur “harten Synchronisation”
  - `cli` und `sti`
  - Unterbrechungsebenen
- Software-Mechanismen zur “weichen Synchronisation”
  - Pro-/Epilogmodell und Varianten
  - Unterbrechungstransparente Algorithmen

VL$_7$ Das x86 Programmiermodell

- Die Entwicklung der x86 CPU-Familie
  - vom 8086 bis zum Core i7
- Relikte und Eigenarten (*quirks*)
  - *Real Mode*
  - *A20 Gate*
- Neuerungen des *Protected Mode*
  - Ringe und Schutzmodell
  - *Task*-Modell
- Hardwarevirtualisierung

VL_8 Programmfäden

- Realisierung von Programmfäden
  - beim MC68k, Infinion TriCore, Intel x86
  - Fortsetzungen und Koroutinen als Basis
  - Implementierung des Kontextwechsels
- Fadenmodelle
  - leicht vs. schwer vs. federgewichtig vs. . . .
  - Umsetzung in einer Systemfamilie

VL_9 Ablaufplanung

- Kurze Wiederholung und Vertiefung
  - Grundprinzipien
  - Klassifikation
  - neue Strategien
- Beispiele aus der Praxis
  - Windows
  - Linux
  - Scheduling auf Multiprozessor-Systemen
- Herausforderungen beim Betriebssystembau
  - Zusammenspiel Ablaufplanung $\rightleftharpoons$ Unterbrechungssynchronisation

VL_{10} Architekturen

- Wie organisiert man ein Betriebssystem: Architekturmodelle
  - Bibliotheken
  - Monolithen
  - Mikrokerne
  - Exokerne
  - Hypervisor
- Geschichte: Revolutionen, Religionen . . . und die Realität
  - Bewertungskriterien
  - Erfolgs- und Misserfolgsgeschichten
- Beispiele aus der Praxis
  - OS360, Unix, Linux, L4, Windows
  - exoKernel, xen, vmware
  - . . .

VL_{11} Fadensynchronisation

- Grundsätzliches
  - Vorraussetzungen
  - aktives und passives Warten
- Synchronisationsprimitiven
  - *Mutex*, *Semaphore* und *Condition*
  - aus der Sicht des BS-Entwicklers
- spezielle Probleme
  - Wechselwirkung Synchronisation $\rightleftharpoons$ Ablaufplanung
  - Fortschrittsgarantie und Verklemmung
- Beispiele aus der Praxis
  - Synchronisationsprimitiven in Windows

VL_{12} Geräteprogrammierung

- Treiber und ihre Bedeutung
  - Vielfalt von Geräten
  - Probleme
- Komponentenmodell für Treiber
  - Struktur eines E/A-Systems
  - Treiberklassen und -schnittstellen
- Beispiele aus der Praxis
  - Windows
  - Linux

VL_{13} Interprozesskommunikation (IPC)

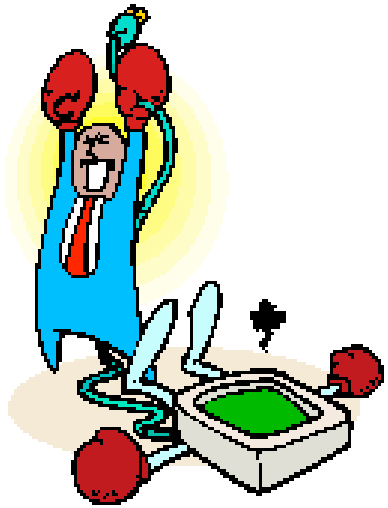
- Grundsätzliches
  - Wechselwirkung $\rightleftharpoons$ Synchronisation
  - implizite und explizite Synchronisation
- Abstraktionen jenseits von *Semaphore*
  - gemeinsamer und verteilter Speicher
  - Fern- und Nahaufrufe
- Dualität nachrichtenbasierter und prozeduraler Systeme
  - konkrete Beispiele
  - Mikrokern $\rightleftharpoons$ Monolith

VL_{14} Zusammenfassung und Ausblick

Quo Vadis Betriebssysteme?

- Zusammenfassung
  - Zusammenfassung des Lernstoffes
  - Diskussion der Evaluationsergebnisse
  - Tipps und Hinweise für die Prüfung
- Ausblick: Neue Herausforderungen
  - Multi- und Manycore Systeme
  - Heterogene Hardware
- Ausblick: Systeme aus der Forschung
  - Corey
  - Barrelfish/Multikernel
  - Factored OS
  - TxOS
  - OctoPOS/Invasive Computing
  - ...

Das Endziel. . .



Viel Spaß!

Betriebssysteme (BS)

VL 2 – Einstieg in die Betriebssystementwicklung

Volkmar Sieh / **Daniel Lohmann**

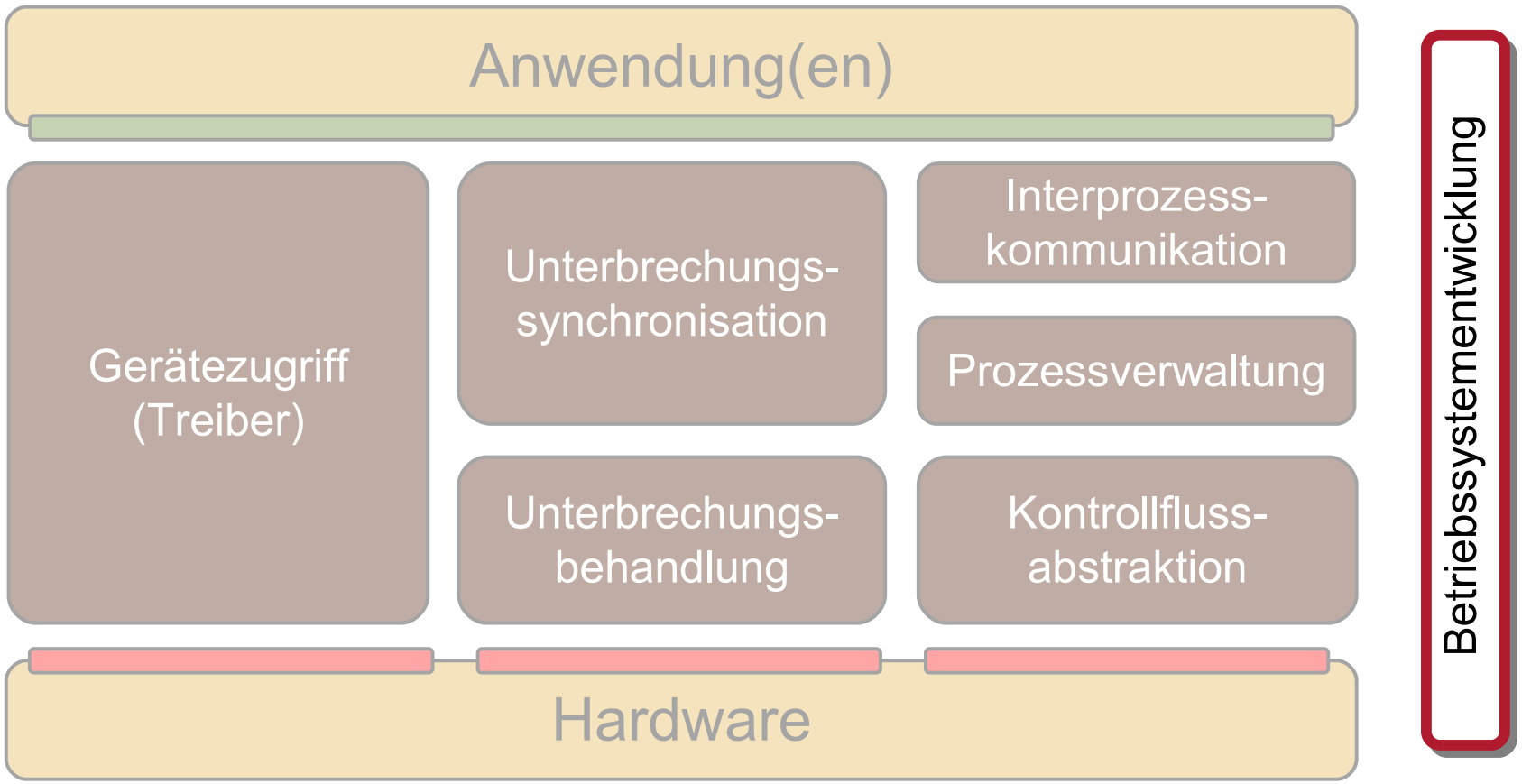
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 25. Oktober 2023

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Agenda

Einordnung
Übersetzen und Linken
Booten
Debugging
Zusammenfassung

Agenda

Einordnung
Übersetzen und Linken
Booten
Debugging
Zusammenfassung

BS-Entwicklung (oft ein harter Kampf)

- Erste Schritte
  wie bringt man sein System auf die Zielhardware?
  - Übersetzung
  - Bootvorgang
- Testen und *Debugging*
  was tun, wenn das System nicht reagiert?
  - „printf“ *debugging*
  - Emulatoren
  - *Debugger*
  - *Remote-Debugger*
  - Hardwareunterstützung

Agenda

Einordnung
Übersetzen und Linken
Booten
Debugging
Zusammenfassung

Übersetzung – *Hello, World?*

```
#include <iostream>

int main () {
  std::cout << “Hello, World“ << std::endl;
}
```

```
> g++ -o hello hello.cc
```

- Annahme:
  - das Entwicklungssystem läuft unter Linux/x86
  - das Zielsystem ist ebenfalls ein PC
- Läuft dieses Programm auch auf der „nackten“ Hardware?
- Kann man Betriebssysteme überhaupt in einer Hochsprache entwickeln?

Übersetzung – Probleme u. Lösungen

- kein dynamischer Binder vorhanden

  ➔ alle nötigen **Bibliotheken statisch einbinden**.

- libstdc++ und libc benutzen Linux Systemaufrufe
  (insbesondere `write`)

  ➔ die normalen C/C++ **Laufzeitbibliotheken können nicht benutzt werden**. Andere haben wir (meistens) nicht.

- generierte Adressen beziehen sich auf virtuellen Speicher!
  ("nm hello | grep main" liefert "0804846c T main")

  ➔ die Standardeinstellungen des Binders können nicht benutzt werden. **Man benötigt eine eigene Binderkonfiguration**.

- der Hochsprachencode stellt Anforderungen
  (Registerbelegung, Adressabbildung, Laufzeitumgebung, Stapel, ...)

  ➔ ein eigener **Startup-Code** (in Assembler erstellt) muss die Ausführung des Hochsprachencodes vorbereiten

Agenda

Einordnung
Übersetzen und Linken
Booten
Debugging
Zusammenfassung

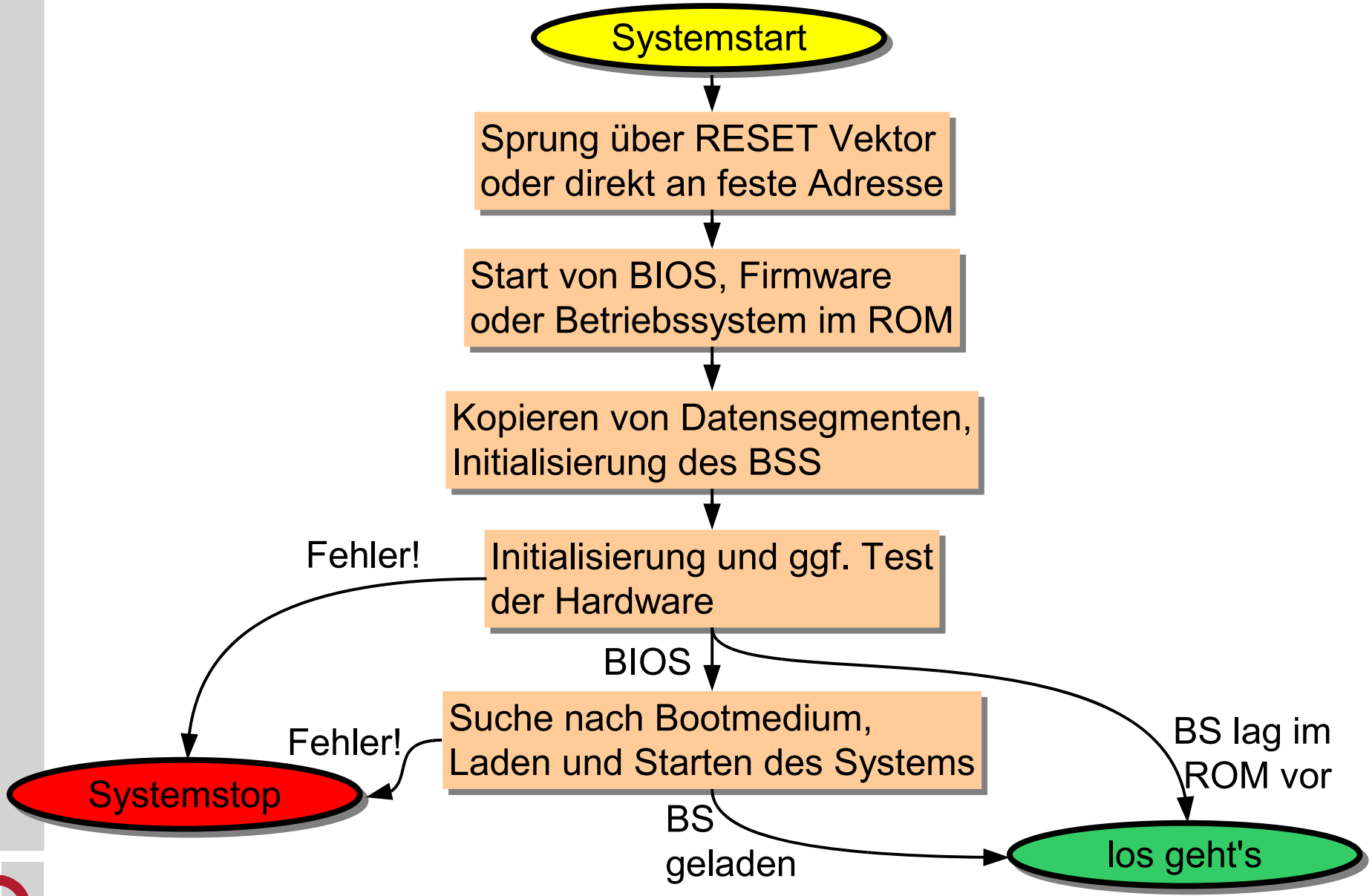
Bootvorgang

*„**Bootstrapping** (englisches Wort für Stiefelschlaufe) bezeichnet einen Vorgang bei dem ein einfaches System ein komplexeres System startet. Der Name des Verfahrens kommt von der Münchhausen-Methode.“*

*„Die **Münchhausen-Methode** bezeichnet allgemein, dass ein System sich selbst in Gang setzt. Die Bezeichnung spielt auf die deutsche Legende von Baron Münchhausen an, der sich an seinen eigenen Haaren aus einem Sumpf gezogen haben soll. In der amerikanischen Fassung benutzte er seine Stiefelschlaufen, was die englische Bezeichnung Bootstrapping für diese Methode begründete.“*

wikipedia.de

Bootvorgang



Bootvorgang beim PC – Bootsektor

- das PC BIOS lädt den 1. Block (512 Bytes) des Bootlaufwerks an die Adresse 0x7c00 und springt dorthin (blind!)
- Aufbau des „Bootsektors“:

FAT Diskette
(DOS/Windows)

Offset	Inhalt
0x0000	`jmp boot; nop;` (ebxx90)
0x0003	Systemname und Version
0x000b	Bytes pro Sektor
0x000d	Sektoren pro Cluster
0x000e	reservierte Sektoren (für Boot Record)
0x0010	Anzahl der FATs
0x0011	Anzahl der Stammverzeichniseinträge
0x0013	Anzahl der logischen Sektoren
0x0015	Medium-Deskriptor-Byte
0x0016	Sektoren pro FAT
0x001a	Anzahl der Köpfe
0x001c	Anzahl der verborgenen Sektoren
0x001e	`boot:` `...`
0x01fe	0xaa55

Bootvorgang beim PC – Bootsektor

- das PC BIOS lädt den 1. Block (512 Bytes) des Bootlaufwerks an die Adresse 0x7c00 und springt dorthin
- Aufbau des „Bootsektors“:

Alternative (OOStuBS):

Offset	Inhalt
0x0000	`jmp boot;`
0x0004	Anzahl der Spuren
0x0006	Anzahl der Köpfe
0x0008	Anzahl der Sektoren
0x000a	reservierte Sektoren (Setup-Code)
0x000c	reservierte Sektoren (System)
0x000e	BIOS Gerätecode
0x000f	Startspur der Diskette/Partition
0x0010	Startkopf der Diskette/Partition
0x0011	Startsektor der Diskette/Partition
0x0010	`boot:` `  ...`
0x01fe	**0xaa55**

Wichtig ist eigentlich nur der Start und die „Signatur“ (0xaa55) am Ende. Alles weitere benutzt der ***Boot-Loader***, um das eigentliche System zu laden.

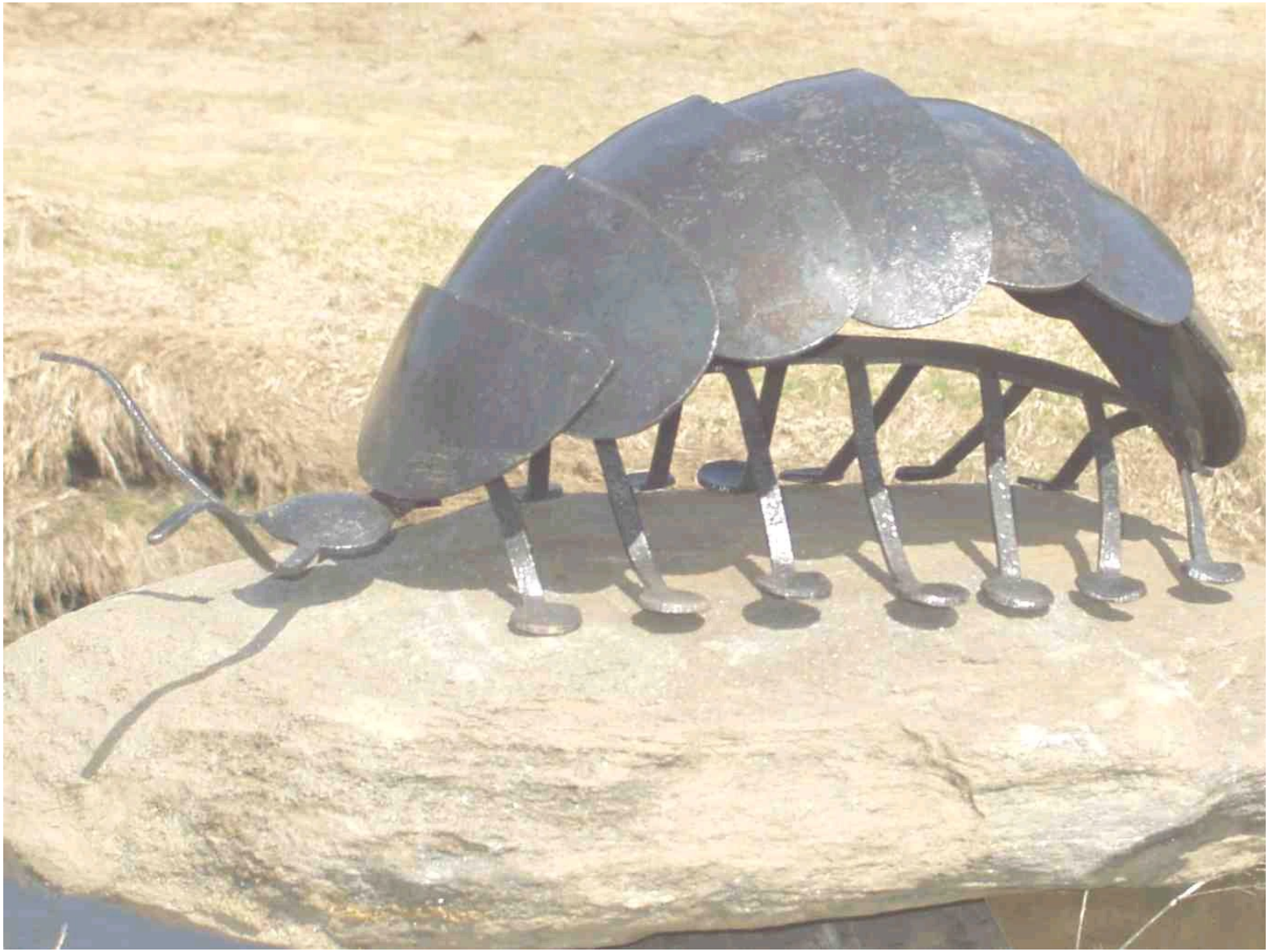
Bootvorgang beim PC – *Boot Loader*

- einfache, systemspezifische *Boot Loader*
  - Herstellung eines definierten Startzustands der Hard- und Software
  - ggf. Laden weiterer Blöcke mit *Boot Loader Code*
  - Lokalisierung des eigentlichen Systems auf dem Boot-Medium
  - Laden des Systems (mittels Funktionen des BIOS)
  - Sprung in das geladene System
- *"Boot Loader"* auf nicht boot-fähigen Disketten
  - Ausgabe einer Fehlermeldung und Neustart
- *Boot Loader* mit Auswahlmöglichkeit
  (z.B. im *Master Boot Record* einer Festplatte)
  - Darstellung eines Auswahlmenüs
  - Nachbildung des BIOS beim Booten des ausgewählten Systems
    - Laden des jeweiligen Bootblocks nach 0x7c00 und Start

Agenda

- Einordnung
- Übersetzen und Linken
- Booten
- Debugging
  - Wie entwanzt man ein BS?
  - „printf”-Debugging
  - Software-Emulatoren
  - Debugger
  - Source-Level-Debugging
  - Remote-Debugging
  - Debugging Deluxe
- Zusammenfassung

Debugging



Der erste dokumentierte „Bug“

9/9

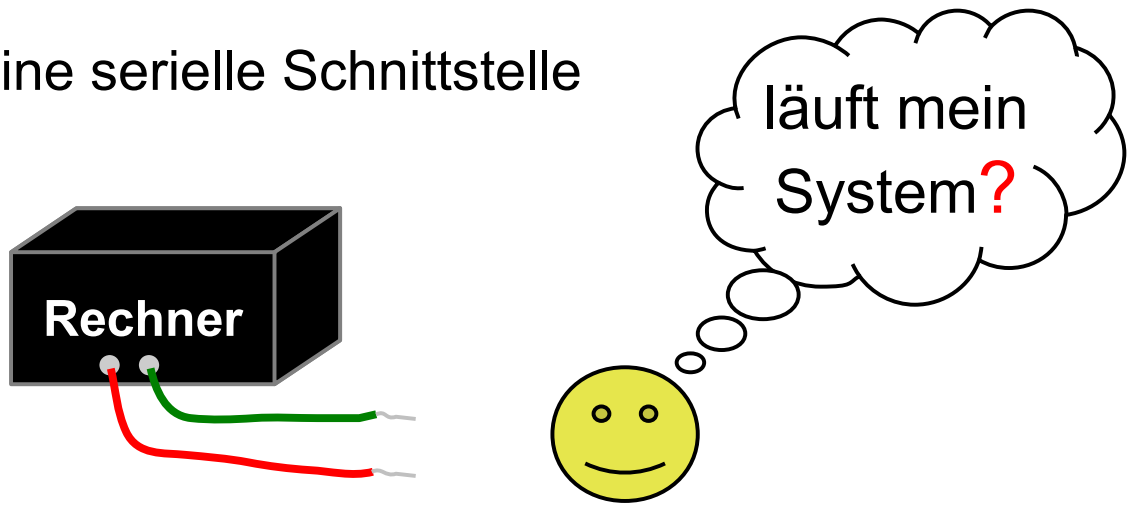
0800 Arctan started
1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 795 correct
13"UC (032) MP - MC 2.130476415 (033) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In Tape " " 10.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545
Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

Quelle: Wikipedia

Admiral Grace Hopper

„printf – Debugging“

- gar nicht so einfach, da es `printf()` per se nicht gibt!
  - oftmals gibt es nicht mal einen Bildschirm
- `printf()` ändert oft auch das Verhalten des *debuggee*
  - mit printf() tritt der Fehler nicht plötzlich nicht mehr / anders auf
  - das gilt gerade auch bei der Betriebssystementwicklung
- Strohhalme
  - eine blinkende LED
  - eine serielle Schnittstelle

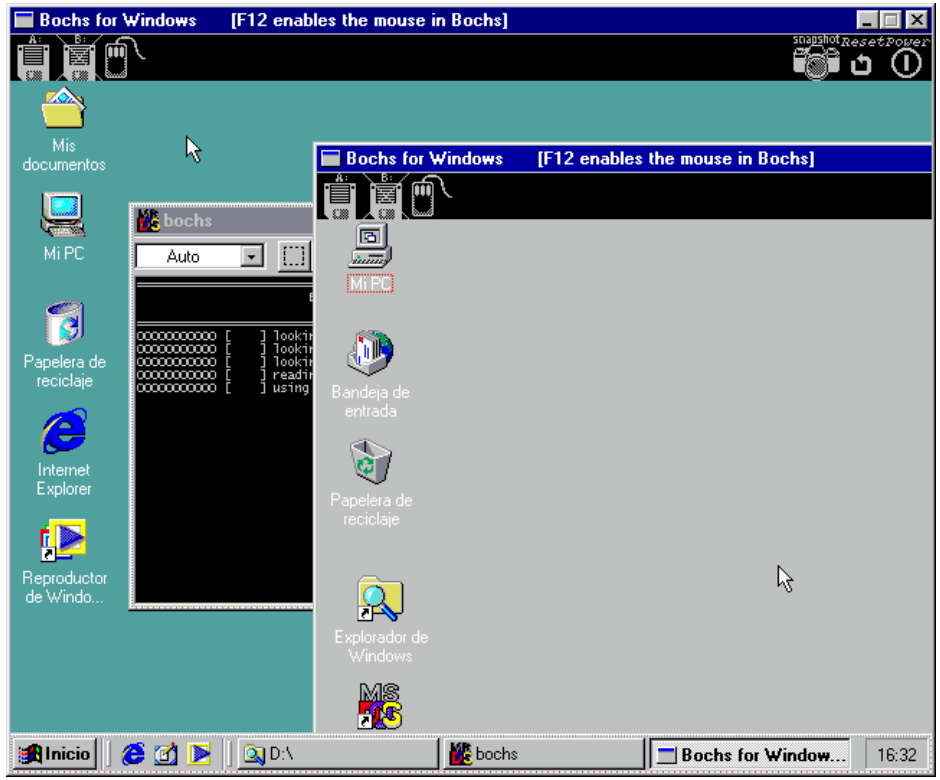


(Software-)Emulatoren

- ahmen reale Hardware in Software nach
  - einfacheres Debugging, da die Emulationssoftware in der Regel kommunikativer als die reale Hardware ist
  - kürzere Entwicklungszyklen
- Vorsicht: am Ende muss das System auf realer Hardware laufen!
  - in Details können sich Emulator und reale Hardware unterscheiden!
  - im fertigen System sind Fehler schwerer zu finden als in einem inkrementell entwickelten System
- übrigens: "virtuelle Maschinen" und "Emulatoren" sind **nicht** gleichbedeutend
  - in VMware wird z.B. kein x86 Prozessor emuliert, sondern ein vorhandener Prozessor führt Maschinencode in der VM direkt aus

Emulatoren – Beispiel "Bochs"

- emuliert i386, ..., Pentium, AMD64 (Interpreter)
  - optional MMX, SSE, SSE2 und 3DNow! Instruktionen
  - Multiprozessoremulation
- emuliert kompletten PC
  - Speicher, Geräte (selbst Sound- und Netzwerkkarte)
  - selbst Windows und Linux Systeme laufen in Bochs
- implementiert in C++
- Entwicklungsunterstützung
  - Protokollinformationen, insbesondere beim Absturz
  - eingebauter Debugger (GDB-Stub)



Bochs in Bochs

Debugging

- ein *Debugger* dient dem Auffinden von Softwarefehlern durch Ablaufverfolgung
  - in Einzelschritten (*single step mode*)
  - zwischen definierten Haltepunkten (*breakpoints*), z.B. bei
    - Erreichen einer bestimmten Instruktion
    - Zugriff auf ein bestimmtes Datenelement
- Vorsicht: manchmal dauert die Fehlersuche mit einem Debugger länger als nötig
  - wer gründlich nachdenkt kommt oft schneller zum Ziel
    - Einzelschritte kosten viel Zeit
    - kein Zurück bei versehentlichem Verpassen der interessanten Stelle
  - beim printf-Debugging können Ausgaben besser aufbereitet werden
  - Fehler im Bereich der Synchronisation nebenläufiger Aktivitäten sind interaktiv mit dem Debugger praktisch nicht zu finden
- praktisch: Analyse von *"core dumps"*
  - beim Betriebssystembau allerdings weniger relevant

Debugging – Beispielsitzung

Setzen eines Abbruchpunktes

Start des Programms

Ablaufverfolgung im Einzelschritt-modus

Fortsetzung des Programms

```
spinczyk@faui48:~> gdb hello
GNU gdb 6.3
...
(gdb) break main
Breakpoint 1 at 0x8048738: file hello.cc, line 5.
(gdb) run
Starting program: hello

Breakpoint 1, main () at hello.cc:5
5           cout << "hello" << endl;
(gdb) next
hello
6           cout << "world" << endl;
(gdb) next
world
7         }
(gdb) continue
Continuing.

Program exited normally.
(gdb) quit
```

Debugging – Funktionsweise (1)

- praktisch alle CPUs unterstützen das *Debugging*
- Beispiel: Intels x86 CPUs
  - die **INT3** Instruktion löst *"breakpoint interrupt"* aus (ein *TRAP*)
    - wird gezielt durch den *Debugger* im Code platziert
    - der *TRAP-Handler* leitet den Kontrollfluss in den *Debugger*
  - durch Setzen des ***Trap Flags* (TF)** im Statusregister (EFLAGS) wird nach **jeder** Instruktion ein *"debug interrupt"* ausgelöst
    - kann für die Implementierung des Einzelschrittmodus genutzt werden
    - der *TRAP-Handler* wird nicht im Einzelschrittmodus ausgeführt
  - mit Hilfe der ***Debug* Register DR0-DR7** (ab i386) können bis zu vier Haltepunkte überwacht werden, ohne den Code manipulieren zu müssen
    - erheblicher Vorteil bei Code im ROM/FLASH oder nicht-schreibbaren Speichersegmenten

→ nächste Folie

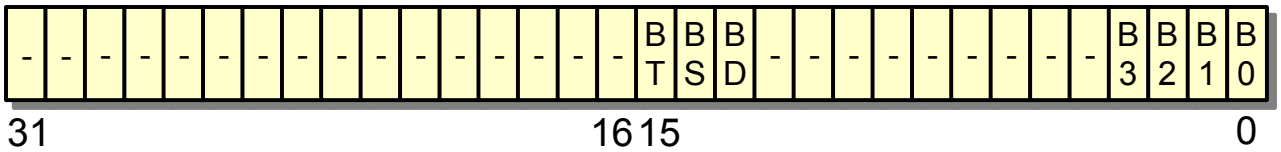
Debugging – Funktionsweise (2)

die Debug Register des 80386

Breakpoint Register

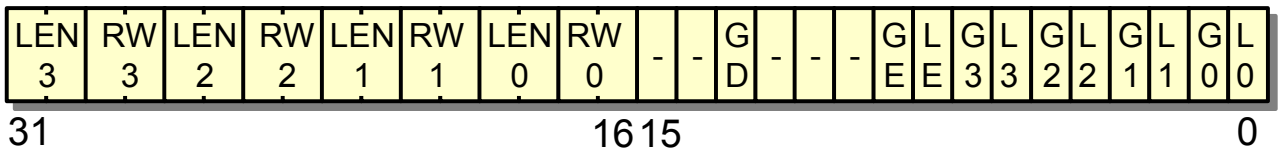
breakpoint 0: lineare Adresse	DR0
breakpoint 1: lineare Adresse	DR1
breakpoint 2: lineare Adresse	DR2
breakpoint 3: lineare Adresse	DR3
reserviert	DR4
reserviert	DR5

Debug Statusregister



DR6

Debug Steuerregister



DR7

Debugging – Funktionsweise (2)

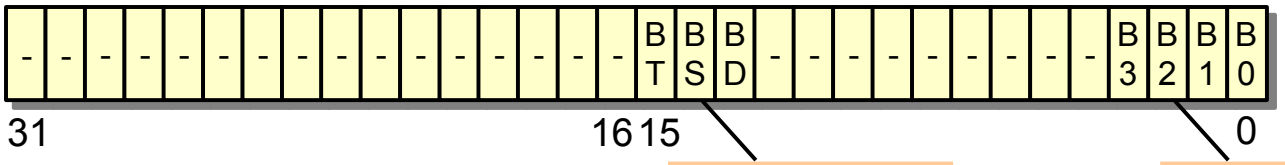
die Debug Register des 80386

Breakpoint Register

breakpoint 0: lineare Adresse	DR0
breakpoint 1: lineare Adresse	DR1
breakpoint 2: lineare Adresse	DR2
breakpoint 3: lineare Adresse	DR3
reserviert	DR4
reserviert	DR5

gibt dem *Trap Handler* Auskunft über die Ursache des *Traps*

Debug Statusregister

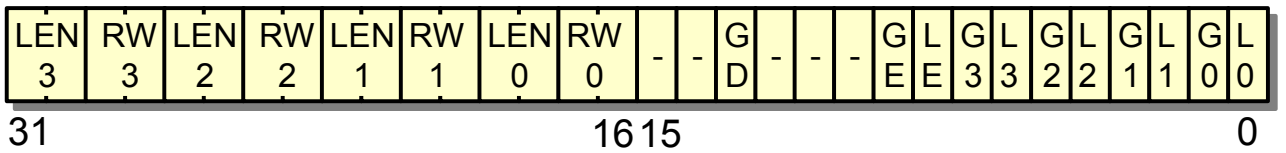


DR6

Einzelschritt

Breakpoint 0-3

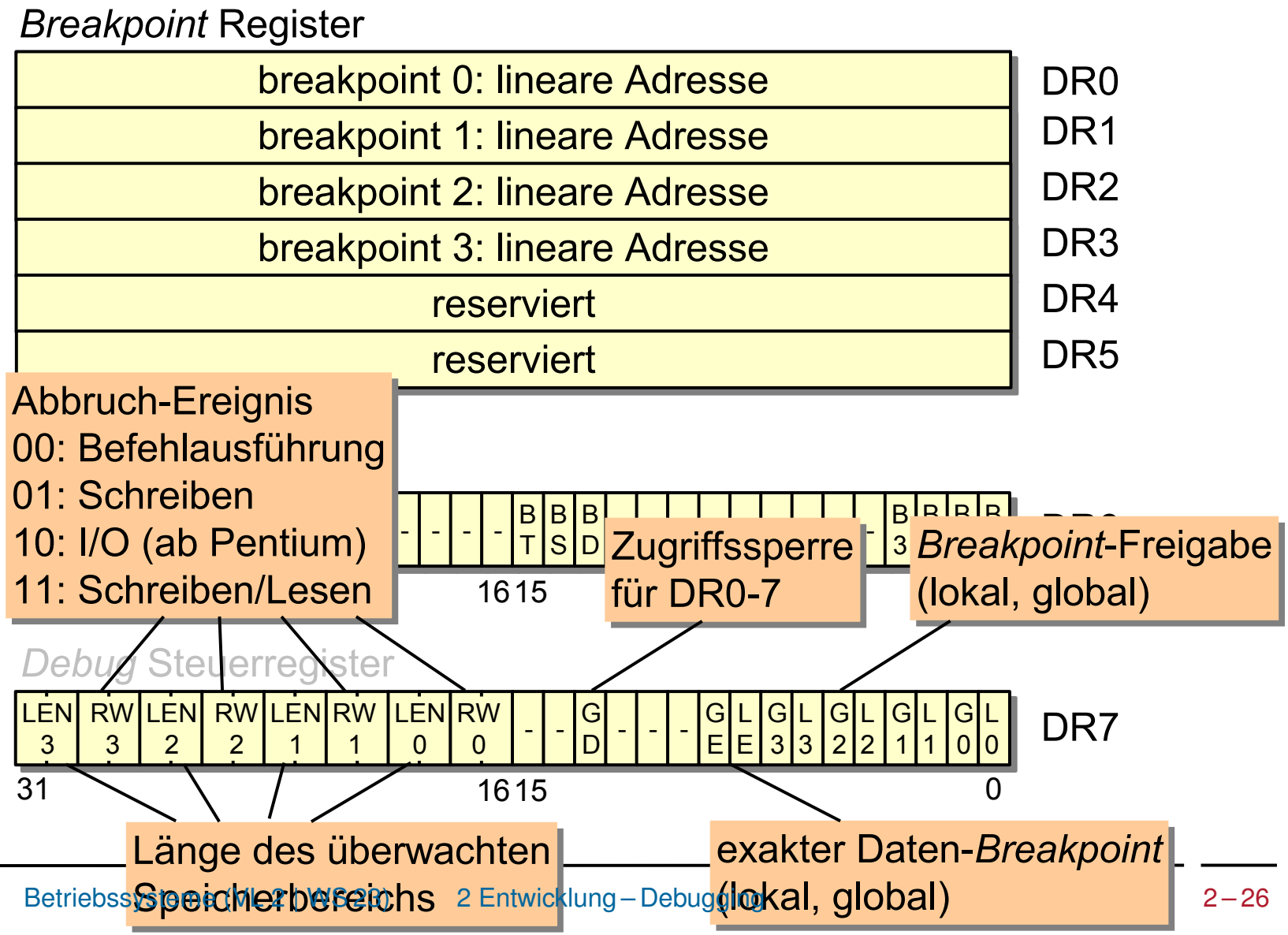
Debug Steuerregister



DR7

Debugging – Funktionsweise (2)

die Debug Register des 80386



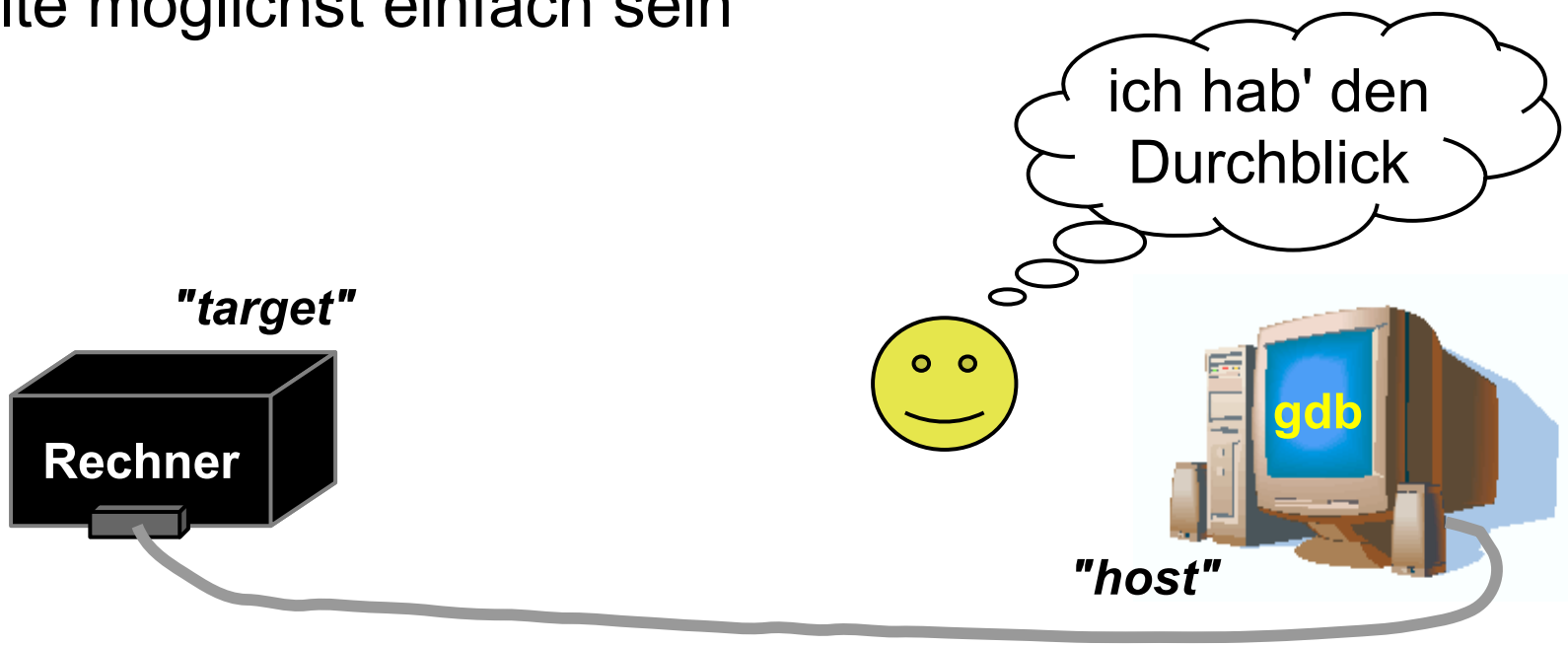
Debugging – Funktionsweise (3)

- besonders effektiv wird Debugging, wenn das Programm im Quelltext visualisiert wird (*source-level debugging*)
  - erfordert Zugriff auf den Quellcode und Debug-Informationen
  - muss durch den Übersetzer unterstützt werden

```
lohmann@faui48:~> g++ -o hello -g hello.cc
lohmann@faui48:~> objdump --section-headers hello
hello:     file format elf32-i386
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
...
 26 .debug_aranges 00000098  00000000  00000000  00000ca0  2**3
                  CONTENTS, READONLY, DEBUGGING
 27 .debug_pubnames 00000100  00000000  00000000  00000d38  2**0
                  CONTENTS, READONLY, DEBUGGING
 28 .debug_info   000032b8  00000000  00000000  00000e38  2**0
                  CONTENTS, READONLY, DEBUGGING
 29 .debug_abbrev 00000474  00000000  00000000  000040f0  2**0
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_line   000003ac  00000000  00000000  00004564  2**0
                  CONTENTS, READONLY, DEBUGGING
 31 .debug_frame  0000008c  00000000  00000000  00004910  2**2
                  CONTENTS, READONLY, DEBUGGING
 32 .debug_str    000001c7  00000000  00000000  0000499c  2**0
                  CONTENTS, READONLY, DEBUGGING
lohmann@faui48:~>
```

Remote Debugging

- bietet die Möglichkeit Programme auf Plattformen zu *debuggen*, die (noch) kein interaktives Arbeiten erlauben
  - setzt eine Kommunikationsverbindung voraus (seriell, Ethernet, ...)
  - erfordert einen Gerätetreiber
  - der Zielrechner kann auch ein Emulator sein (z.B. Bochs)
- die *Debugging*-Komponente auf dem Zielsystem (*stub*) sollte möglichst einfach sein



Remote Debugging – Beispiel gdb (1)

- das Kommunikationsprotokoll
  ("GDB *Remote Serial Protocol*" - RSP)
  - spiegelt die Anforderungen an den gdb *stub* wieder
  - basiert auf der Übertragung von ASCII Zeichenketten
  - Nachrichtenformat: **$**<*Kommando oder Antwort*>**#**<*Prüfsumme*>
  - Nachrichten werden unmittelbar mit **+** (OK) oder **–** (Fehler) beantwortet
- Beispiele:
  - **$**g**#**67 ►Lesen aller Registerinhalte
    - Antwort: **+** **$**123456789abcdef0...**#**... ► Reg. 1 ist 0x12345678, 2 ist 0x9...
  - **$**G123456789abcdef0...**#**... ► Setze Registerinhalte
    - Antwort: **+** **$**OK**#**9a ► hat funktioniert
  - **$**m4015bc,2**#**5a ► Lese 2 Bytes ab Adresse 0x4015bc
    - Antwort: **+** **$**2f86**#**06 ► Wert ist 0x2f86

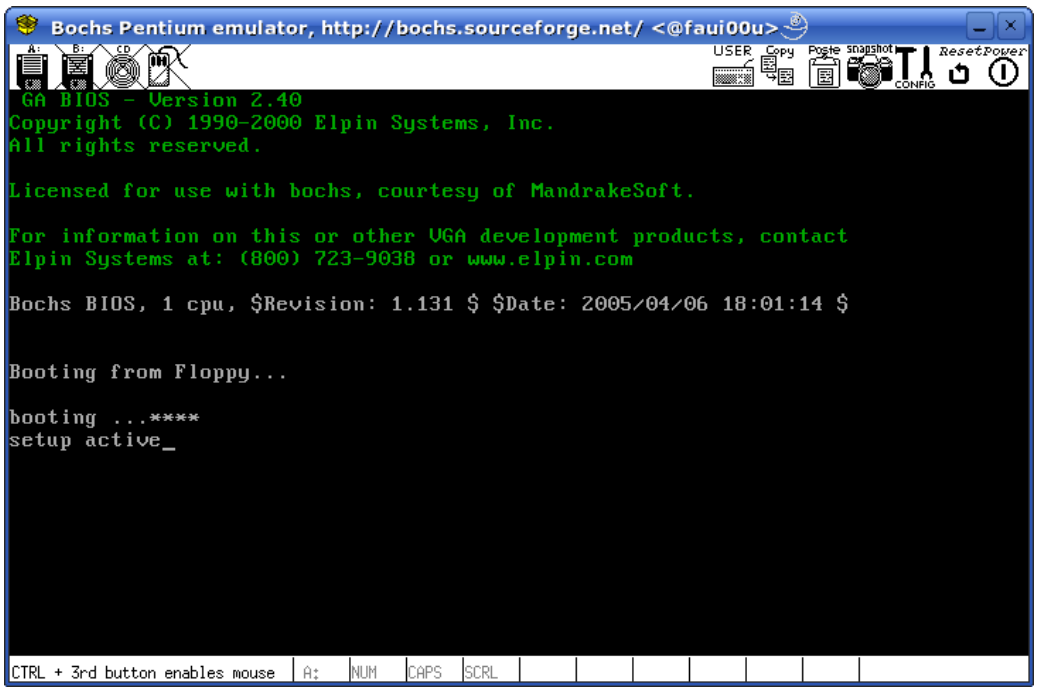
Remote Debugging – Beispiel gdb (2)

- das Kommunikationsprotokoll – kompletter Umfang
  - Register- und Speicherbefehle
    - lese/schreibe alle Register
    - lese/schreibe einzelnes Register
    - lese/schreibe Speicherbereich
  - Steuerung der Programmausführung
    - letzte Unterbrechungsursache abfragen
    - Einzelschritt
    - mit Ausführung fortfahren
  - Sonstiges
    - Ausgabe auf der *Debug* Konsole
    - Fehlernachrichten
- allein "schreibe einzelnes Register", "lese/schreibe Speicherbereich" und "mit Ausführung fortfahren" müssen notwendigerweise vom *stub* implementiert werden

Remote Debugging – mit Bochs

- durch geeignete Konfigurierung vor der Übersetzung kann der Emulator Bochs auch einen gdb *stub* implementieren

```
> bochs-gdb build/bootdisk.img
...
Waiting for gdb connection on
localhost:10452
```



Remote Debugging – mit Bochs

```
> gdb build/system
GNU gdb 6.3-debian
...
(gdb) break main
Breakpoint 1 at 0x11fd8: file main.cc, line 38.
(gdb) target remote localhost:10452
Remote debugging using localhost:10452
0x0000fff0 in ?? ()
(gdb) continue
Continuing.

Breakpoint 1, main () at main.cc:38
38          Application application(appl_stack+sizeof(appl_stack));
(gdb) next
43          for (y=0; y<25; y++)
(gdb) next
44             for (x=0; x<80; x++)
(gdb) next
45                kout.show (x, y, ' ', CGA_Screen::STD_ATTR);
(gdb) continue
Continuing.
```

Debugging Deluxe

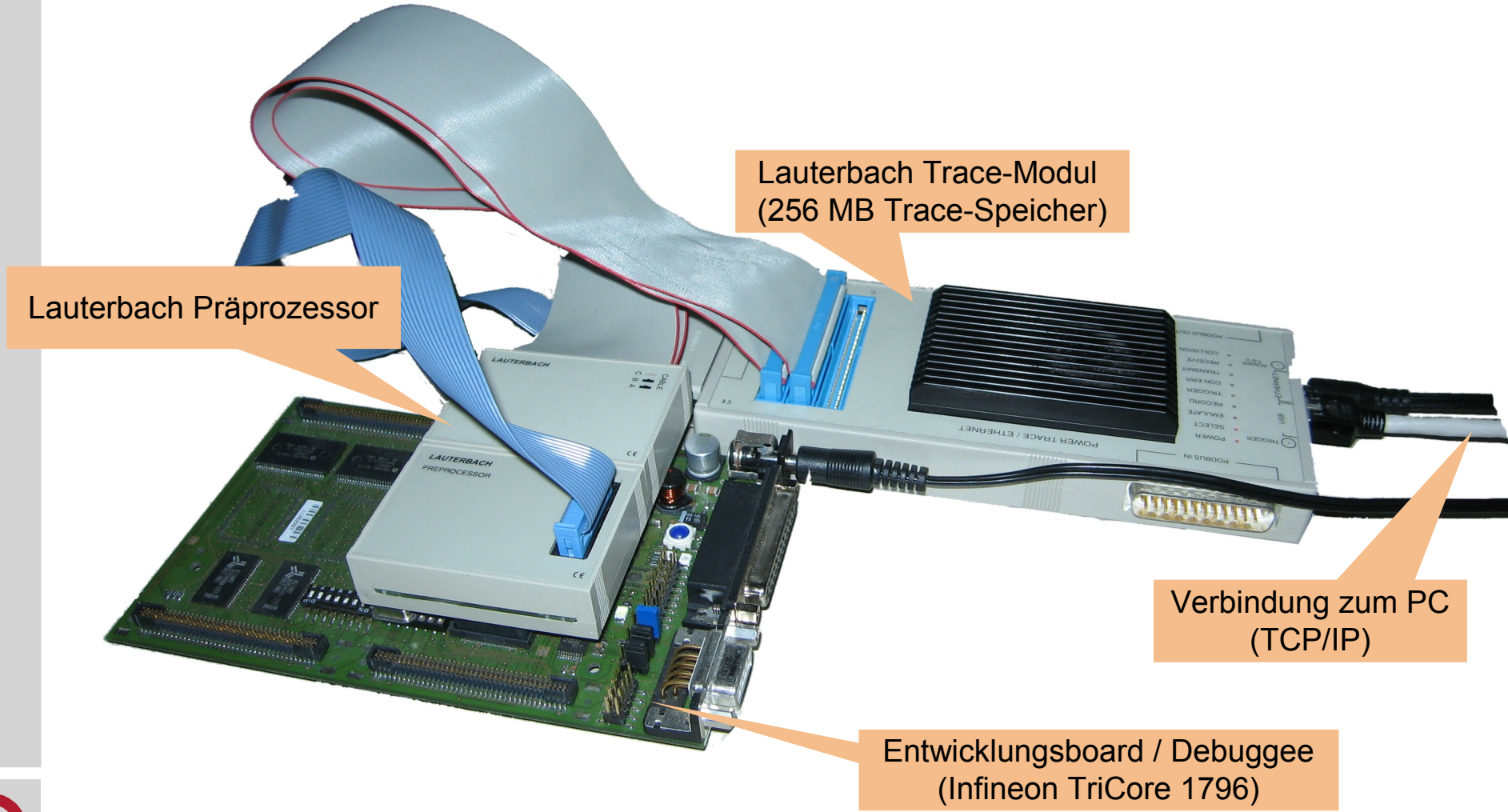
- viele Prozessorhersteller integrieren heute Hardwareunterstützung für *Debugging* auf ihren Chips (*OCDS – On Chip Debug System*)
  - BDM, OnCE, MPD, JTAG
- i.d.R. einfaches serielles Protokoll zwischen *Debugging*-Einheit und externem *Debugger* (Pins sparen!)
- Vorteile:
  - der *Debug Monitor* (z.B. gdb *stub*) belegt keinen Speicher
  - Implementierung eines *Debug Monitors* entfällt
  - Haltepunkte im ROM/FLASH durch Hardware-Breakpoints
  - Nebenläufiger Zugriff auf Speicher und CPU Register
  - mittels Zusatzhardware ist zum Teil auch das Aufzeichnen des Kontrollflusses zwecks nachträglicher Analyse möglich

Debugging Deluxe – Beispiel BDM

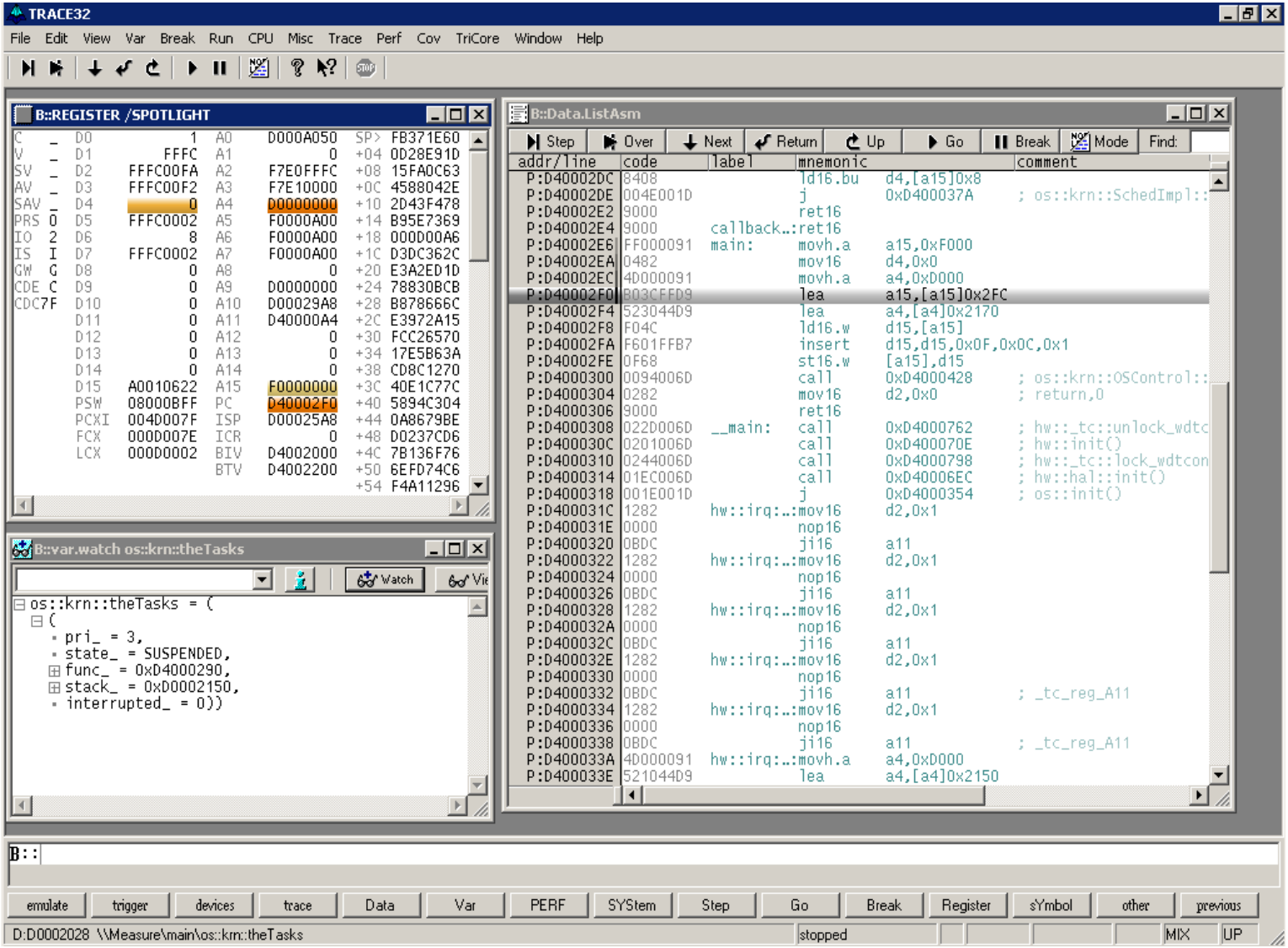
- *"Background Debug Mode"* - eine *on-chip debug* Lösung von Motorola
- serielle Kommunikation über drei Leitungen (DSI, DSO, DSCLK)
- BDM Kommandos der 68k und ColdFire Prozessoren
  - RAREG/RDREG – Read Register
    - lese bestimmtes Daten- oder Adressregister
  - WAREG/WDREG – Write Register
    - schreibe bestimmtes Daten- oder Adressregister
  - READ/WRITE – Read Memory/Write Memory
    - lese/schreibe eine bestimmte Speicherstelle
  - DUMP/FILL – Dump Memory/Fill Memory
    - lese/fülle einen ganzen Speicherblock
  - BGND/GO – Enter BDM/Resume
    - Ausführung stoppen/wieder aufnehmen

Debugger Deluxe: Hardware-Lösung

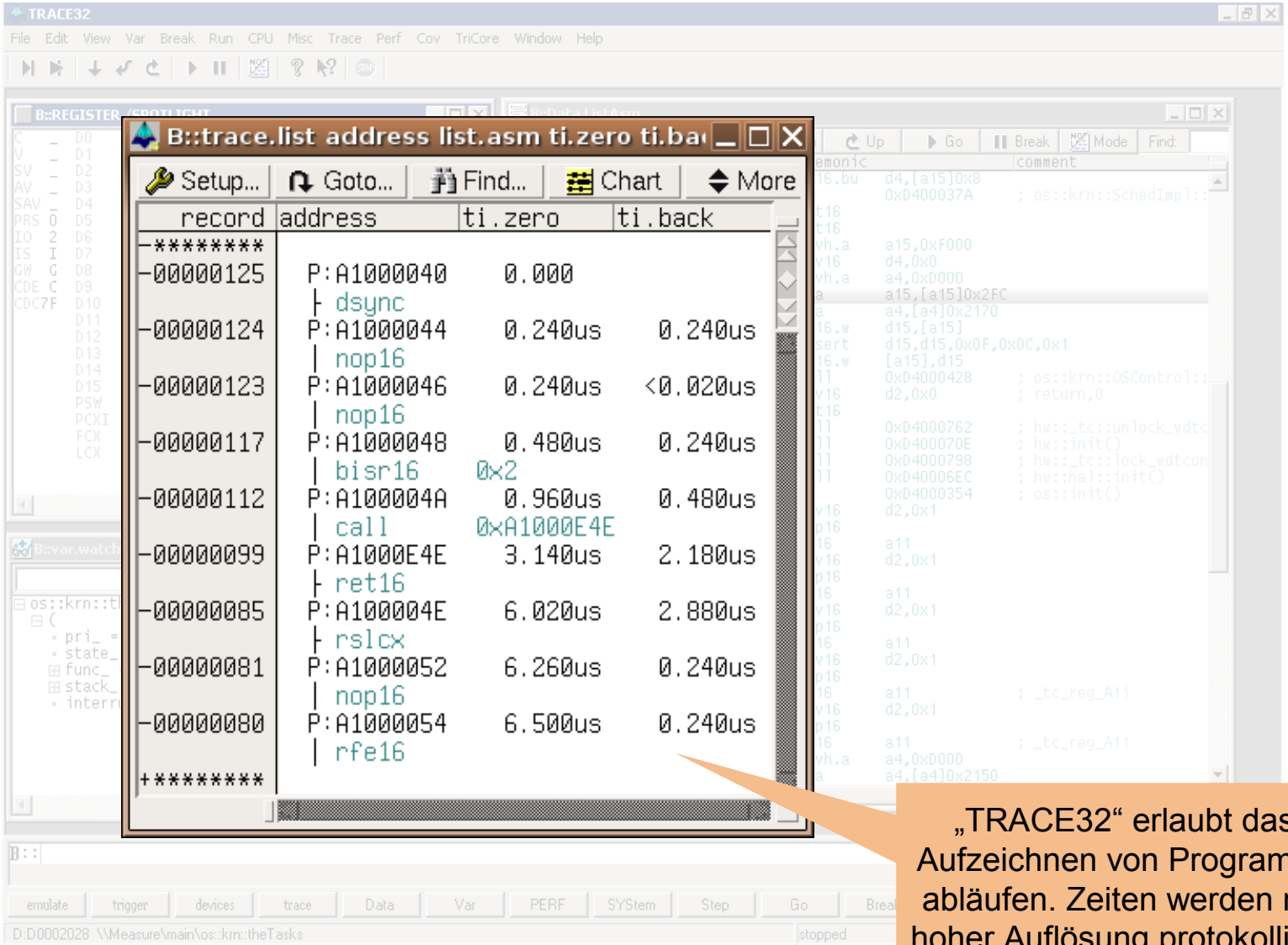
- Lauterbach Hardware-Debugger



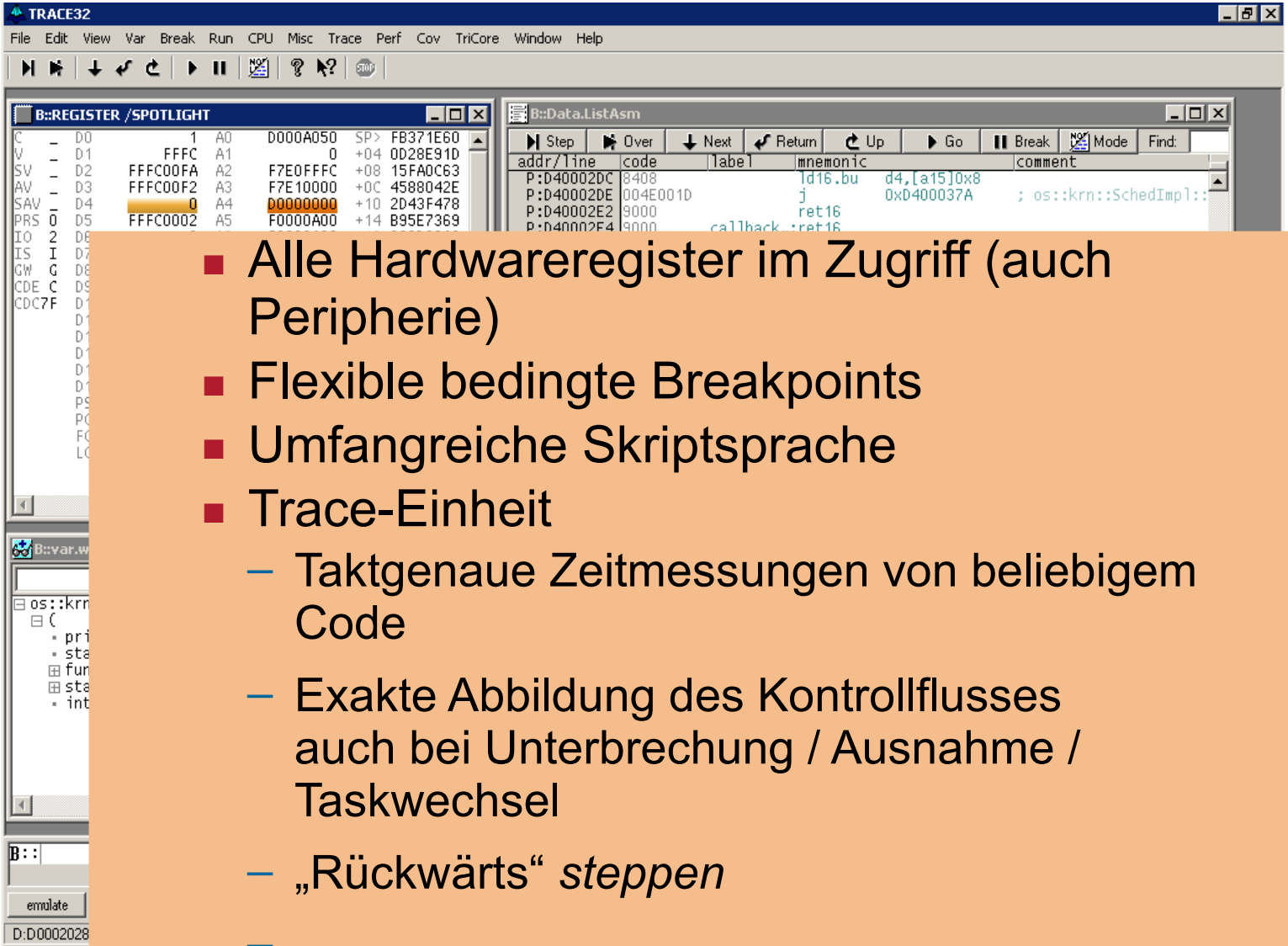
Debugger Deluxe: Lauterbach-Frontend



Debugger Deluxe: Lauterbach-Frontend



Debugger Deluxe: Lauterbach-Frontend



- Alle Hardwareregister im Zugriff (auch Peripherie)
- Flexible bedingte Breakpoints
- Umfangreiche Skriptsprache
- Trace-Einheit
  - Taktgenaue Zeitmessungen von beliebigem Code
  - Exakte Abbildung des Kontrollflusses auch bei Unterbrechung / Ausnahme / Taskwechsel
  - „Rückwärts“ *steppen*
  - *...*

Agenda

Einordnung
Übersetzen und Linken
Booten
Debugging
Zusammenfassung

Zusammenfassung

- Betriebssystementwicklung unterscheidet sich deutlich von gewöhnlicher Applikationsentwicklung:
  - Bibliotheken fehlen
  - die „nackte“ Hardware bildet die Grundlage
- die ersten Schritte sind oft die schwersten
  - Übersetzung
  - Bootvorgang
  - Systeminitialisierung
- komfortable Fehlersuche erfordert eine Infrastruktur
  - Gerätetreiber für printf-*Debugging*
  - STUB und Verbindung/Treiber für *Remote Debugging*
  - Hardware Debugging-Unterstützung wie mit BDM
  - Optimal: Hardware-Debugger wie Lauterbach

Betriebssysteme (BS)

VL 3 – Unterbrechungen, Hardware

Volkmar Sieh / **Daniel Lohmann**

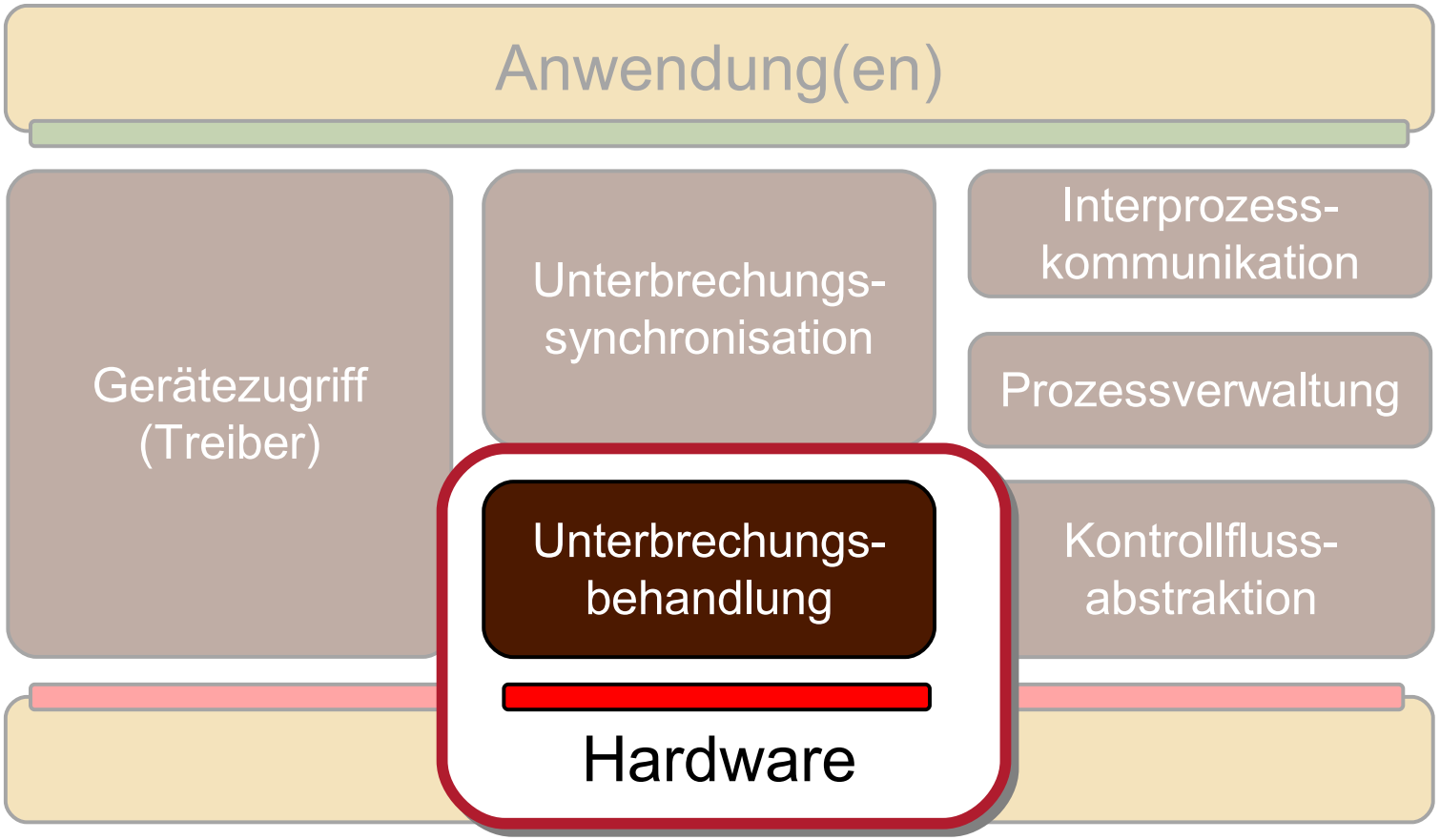
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 8. November 2023

`https://sys.cs.fau.de/lehre/ws23/bs`

Überblick: Einordnung dieser VL



Agenda

Einordnung
Grundlagen
Hardware-Architekturen
Zusammenfassung

Agenda

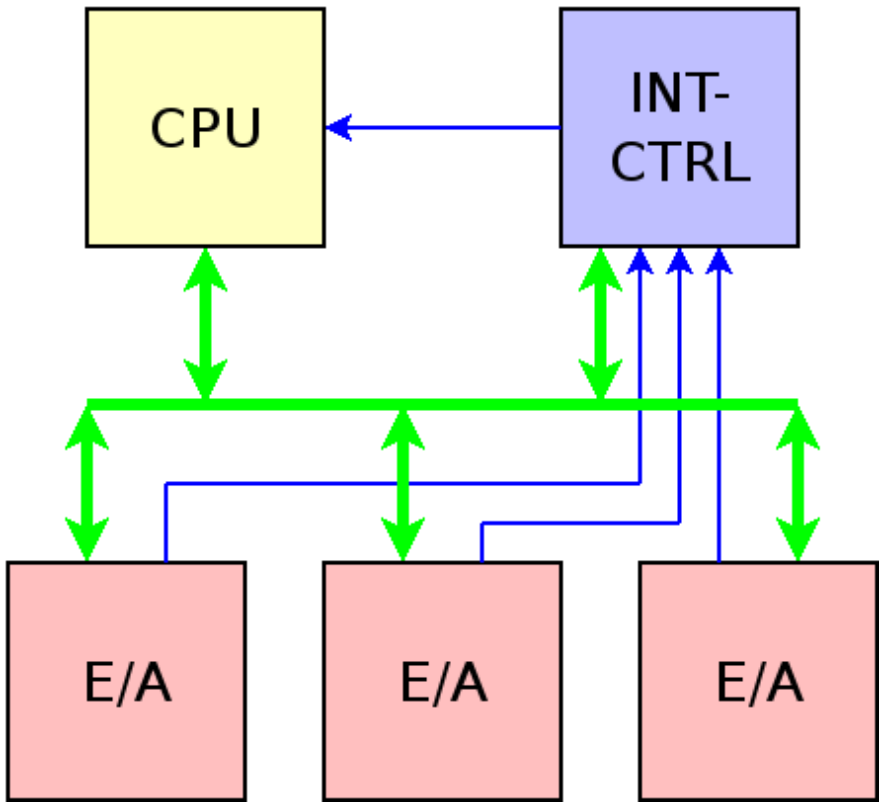
Einordnung
Grundlagen
Hardware-Architekturen
Zusammenfassung

Sinn und Zweck von Unterbrechungen

Ein Blick zurück in die Historie von Betriebssystemen...

- Überlappte Ein-/Ausgabe:
  - Eingaben: Verschwendung von anderweitig nutzbaren Prozessorzyklen bei (oft nicht vorhersagbar langem) *aktivem Warten*
  - Ausgaben: selbständiges Agieren der E/A-Geräte (z.B. durch *DMA*) entlastet die CPU
- Timesharing Betrieb:
  - Zeitgeber-Unterbrechungen geben dem Betriebssystem die Möglichkeit
    - zur *Verdrängung von Prozessen*
    - Aktivitäten zeitgesteuert zu starten

Interrupt-Hardware



Interrupt-Controller ist im einfachsten Fall ein „Oder”-Gatter...

Agenda

Einordnung

Grundlagen

- Priorisierung
- Verlust von IRQs
- Behandlungsroutine
- Zustandssicherung
- Multiprozessorsysteme
- Gefahren

Hardware-Architekturen

Zusammenfassung

Priorisierung

- **Problem**:
  - Mehrere Unterbrechungsanforderungen können gleichzeitig signalisiert werden. Welche ist wichtiger?
  - Während die CPU auf die wichtigste Anforderung reagiert, können weitere Anforderungen signalisiert werden.

Priorisierung

- **Problem**:
  - Mehrere Unterbrechungsanforderungen können gleichzeitig signalisiert werden. Welche ist wichtiger?
  - Während die CPU auf die wichtigste Anforderung reagiert, können weitere Anforderungen signalisiert werden.
- **Lösung**: ein **Priorisierungsmechanismus** ...
  - **in Software**: die CPU hat nur einen IRQ (*interrupt request*) Eingang und fragt die Geräte in bestimmter Reihenfolge ab
  - **in Hardware**: eine Priorisierungsschaltung ordnet Geräten eine Priorität zu und leitet immer nur die dringendste Anforderung zur Behandlung weiter
  - **mit festen Prioritäten**: jedem Gerät wird statisch eine Priorität zugeordnet
  - **mit variablen Prioritäten**: Prioritäten sind dynamisch änderbar oder wechseln zum Beispiel zyklisch

Verlust von IRQs

- **Problem**:
  - während der Behandlung oder Sperrung von Unterbrechungen, kann die CPU keine neuen Unterbrechungen behandeln
  - die Speicherkapazität für Unterbrechungsanforderungen ist endlich.
    - i.d.R. ein Bit pro Unterbrechungseingang

Verlust von IRQs

- **Problem**:
  - während der Behandlung oder Sperrung von Unterbrechungen, kann die CPU keine neuen Unterbrechungen behandeln
  - die Speicherkapazität für Unterbrechungsanforderungen ist endlich.
    - i.d.R. ein Bit pro Unterbrechungseingang
- **Lösung**: in Software
  - die Unterbrechungsbehandlungsroutine sollte möglichst kurz sein (zeitlich!), um die Wahrscheinlichkeit von Verlusten zu minimieren
  - Unterbrechungen sollten nicht unnötig lange gesperrt werden
  - jeder Gerätetreiber sollte davon ausgehen, dass **eine** Unterbrechung **mehr als eine** abgeschlossene E/A Operation anzeigen kann

Zuordnung einer Behandlungsroutine

- **Problem**:
  - die Software soll mit möglichst wenig Aufwand herausfinden können, welches Gerät die Unterbrechung ausgelöst hat
    - eine sequentielle Abfrage der Geräte kostet nicht nur Zeit, sondern verändert die Zustände von E/A Bussen und unbeteiligten Geräten

Zuordnung einer Behandlungsroutine

- **Problem**:
  - die Software soll mit möglichst wenig Aufwand herausfinden können, welches Gerät die Unterbrechung ausgelöst hat
    - eine sequentielle Abfrage der Geräte kostet nicht nur Zeit, sondern verändert die Zustände von E/A Bussen und unbeteiligten Geräten
- **Lösung**:
  - jeder Unterbrechung wird eine Nummer zugeordnet, die als Index in eine Vektortabelle verwendet wird
    - die Vektornummer hat nicht zwangsläufig etwas mit der Priorität zu tun
    - es kommt in der Praxis leider vor, dass Geräte sich eine Vektornummer teilen müssen (*interrupt sharing*)
  - der Aufbau der Vektortabelle variiert je nach Prozessortyp
    - meist enthält sie Zeiger auf Funktionen
    - seltener sind die Einträge selbst bereits Instruktionen

Zustandssicherung

- **Problem**:
  - nach der Ausführung der Behandlungsroutine soll zum normalen Kontext zurückgekehrt werden können
  - die Behandlung soll quasi unbemerkt ablaufen (*transparency*)

Zustandssicherung

- **Problem**:
  - nach der Ausführung der Behandlungsroutine soll zum normalen Kontext zurückgekehrt werden können
  - die Behandlung soll quasi unbemerkt ablaufen (*transparency*)
- **Lösung**:
  - Zustandssicherung durch Hardware
    - nur das Notwendigste: z.B. Rücksprungadresse u. Prozessorstatuswort
    - Wiederherstellung durch speziellen Befehl, z.B. iret, rte, ...
  - Zustandssicherung durch Software
    - da Unterbrechungen jederzeit auftreten können, muss auch die Behandlungsroutine Zustände sichern und wiederherstellen

Geschachtelte Behandlung

- **Problem**:
  - um auf sehr wichtige Ereignisse schnell reagieren zu können, soll auch eine Unterbrechungsbehandlung unterbrechbar sein
  - eine unbegrenzte Schachtelungstiefe muss aber vermieden werden

Geschachtelte Behandlung

- **Problem**:
  - um auf sehr wichtige Ereignisse schnell reagieren zu können, soll auch eine Unterbrechungsbehandlung unterbrechbar sein
  - eine unbegrenzte Schachtelungstiefe muss aber vermieden werden
- **Lösung**:
  - die CPU erlaubt immer nur Unterbrechungen mit höherer Priorität
  - die aktuelle Priorität wird im Prozessorstatuswort gespeichert
  - die vorherige Priorität wird auf einem Stapel abgelegt

Interrupt-Ablauf

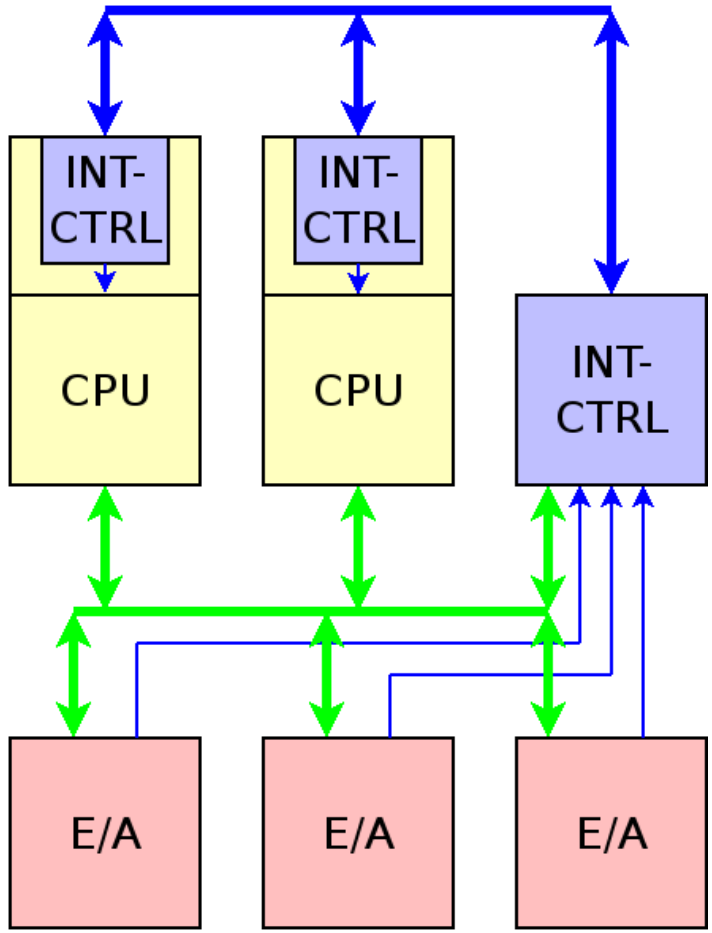
HW: E/A-Gerät signalisiert Interrupt an Interrupt-Controller
HW: Interrupt-Controller signalisiert Interrupt an CPU
HW: CPU sichert einige Register (PC, IE, ...)
HW: CPU disabled Interrupt-Eingang (IE = 0)
HW: CPU holt sich von Interrupt-Controller IRQ-Nummer
HW: CPU lädt Vektor aus Interrupt-Vektor-Tabelle in PC
SW: Interrupt-Handler sichert weitere Register
SW: Interrupt-Handler behandelt Interrupt
SW: Interrupt-Handler restauriert Register
SW: Interrupt-Handler ruft `iret` auf
HW: CPU restauriert PC, IE, ...
Rücksprung in das unterbrochene Haupt-Programm,
Interrupt-Eingang wieder enabled

Multiprozessorsysteme

- **Problem**:
  - Unterbrechungen können immer nur von einer CPU behandelt werden. Aber welche?
  - es gibt eine weitere Kategorie von Unterbrechungen: die Interprozessor-Unterbrechungen
- **Lösung**: die Hardware zur Unterbrechungsbehandlung auf Multiprozessorsystemen muss komplexer ausgelegt sein. Es gibt viele Entwurfsvarianten ...
  - feste Zuordnung
  - zufällige Zuordnung
  - programmierbare Zuordnung
  - Zuordnung unter Berücksichtung der Prozessorlast

  ... und Kombinationen davon.

Interrupt-Hardware (SMP)



Interrupt-Controller sind selbständige, konfigurierbare Einheiten...

Gefahr: „unechte Unterbrechungen“

(„*spurious interrupts*“)

- **Problem**: ein technischer Mechanismus zur Unterbrechungsbehandlung birgt die Gefahr von fehlerhaften Unterbrechungsanforderungen, z.B. durch ...
  - Hardwarefehler
  - fehlerhaft programmierte Geräte
- **Lösung**:
  - Hardware- und Softwarefehler vermeiden
  - Betriebssystem „defensiv“ programmieren
    - mit unechten Unterbrechungen rechnen

Gefahr: „Unterbrechungsstürme“

(„*interrupt storms*“)

- **Problem**:
  - hochfrequente Unterbrechungsanforderungen können einen Rechner lahm legen
  - es handelt sich entweder um unechte Unterbrechungen oder der Rechner ist mit der E/A Last überfordert
  - kann leicht mit Seitenflattern (*thrashing*) verwechselt werden
- **Lösung**: durch das Betriebssystem
  - Unterbrechungsstürme erkennen
  - das verursachende Gerät deaktivieren

Agenda

Einordnung

Grundlagen

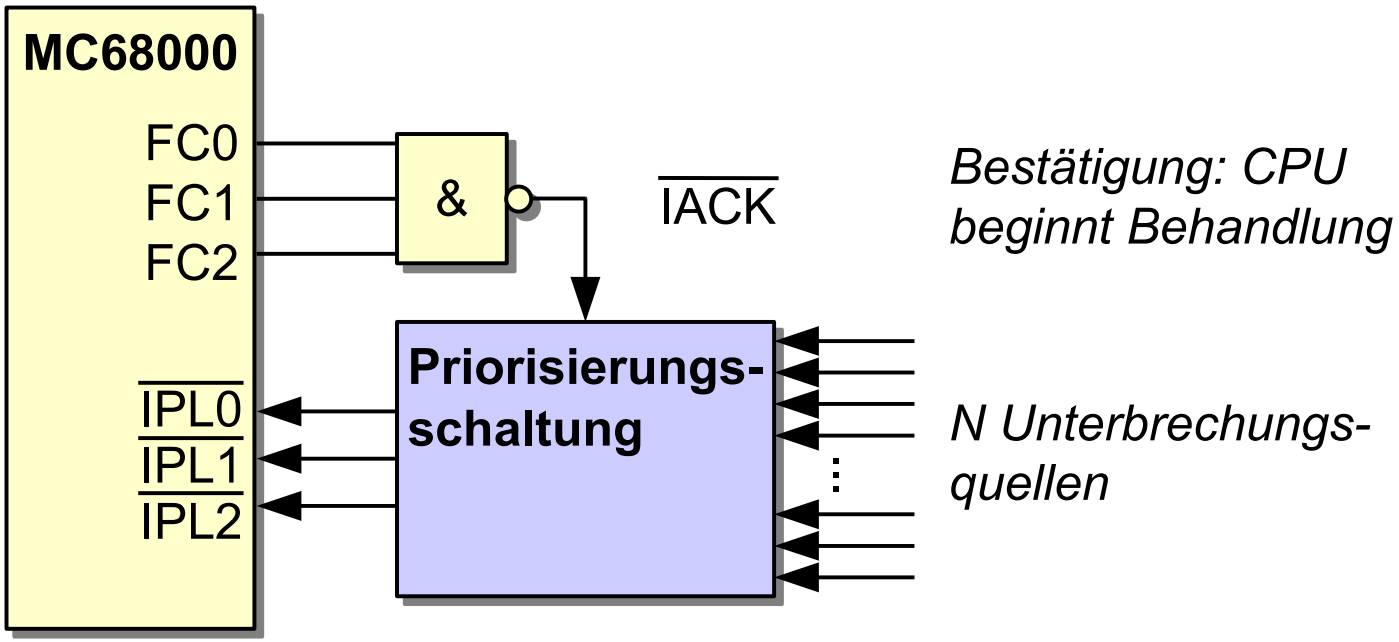
Hardware-Architekturen

- Motorola/Freescale 68k
- Intel x86

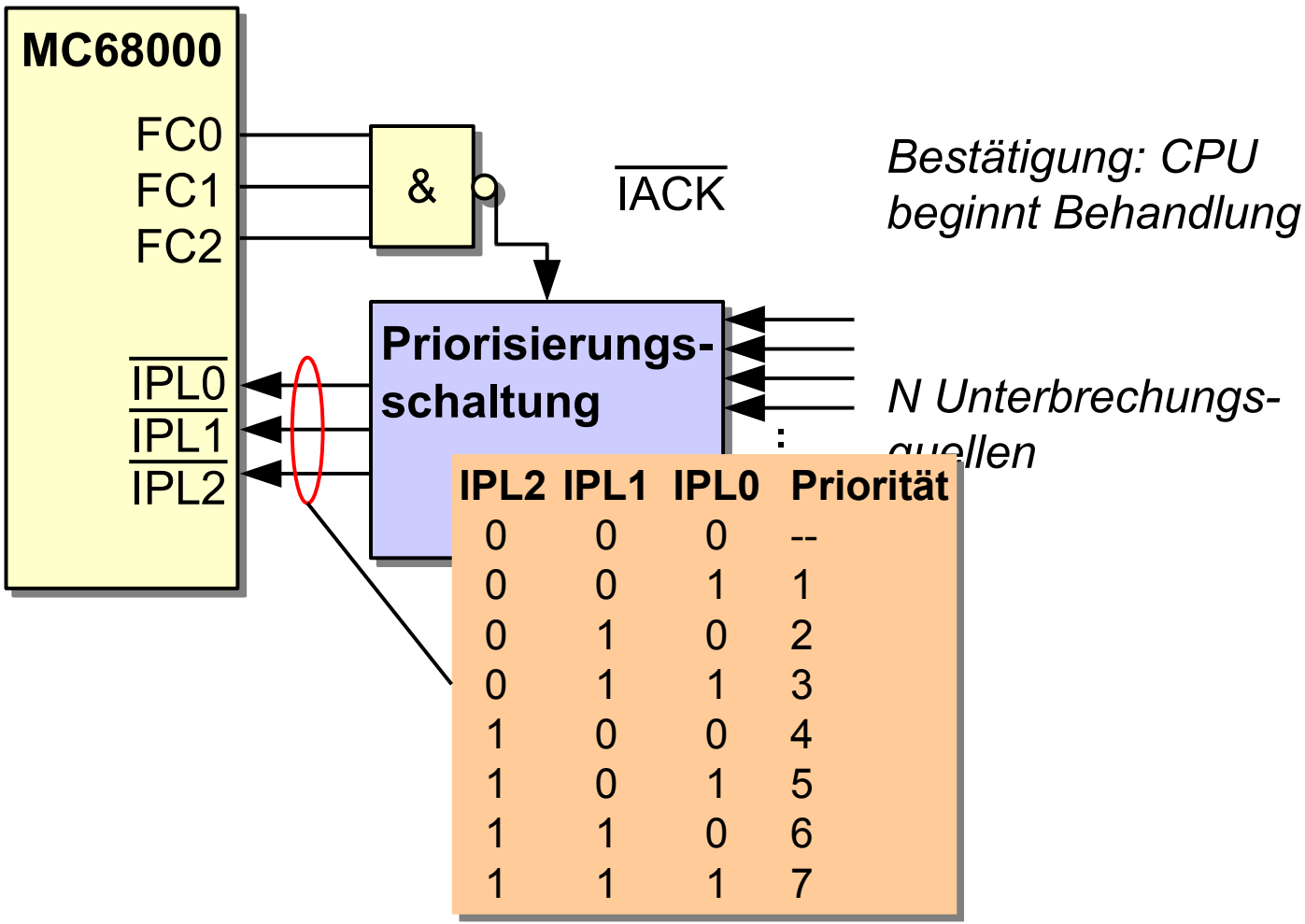
Zusammenfassung

Unterbrechungen beim MC68000

Unterbrechungen beim MC68000

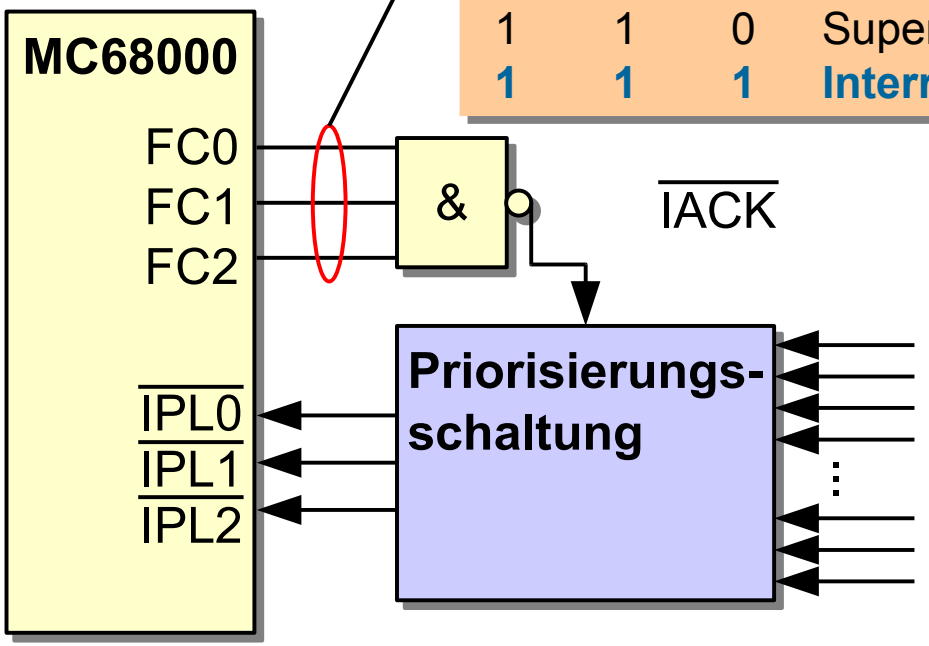


Unterbrechungen beim MC68000



Unterbrechungen

FC2	FC1	FC0	Zyklustyp
0	0	0	reserviert
0	0	1	Anwender-Daten
0	1	0	Anwender-Program
0	1	1	reserviert
1	0	0	reserviert
1	0	1	Supervisor-Daten
1	1	0	Supervisor-Programm
1	**1**	**1**	**Interrupt aufgetreten**

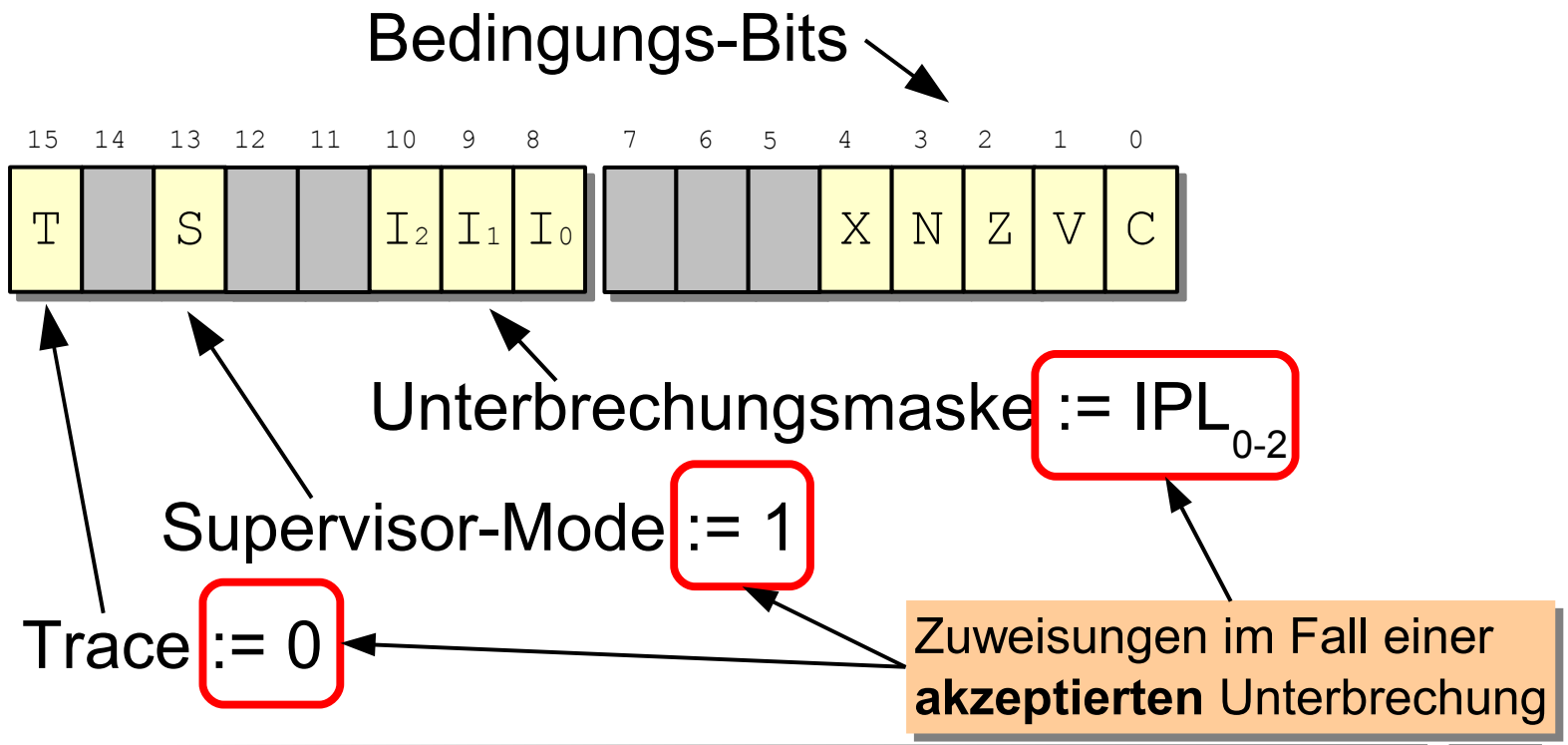


Bestätigung: CPU beginnt Behandlung

N Unterbrechungs-quellen

Das Statusregister (SR) des MC68000

- enthält u.A. die aktuelle Unterbrechungsmaske
  - bei einer Unterbrechung wird geprüft, ob $IPL_{0-2} > I_{0-2}$ ist. Wenn nein, wird der Anforderung (noch) nicht stattgegeben.
  - eine Unterbrechung mit $IPL_{0-2} = 7$ wird aber immer bearbeitet (NMI)



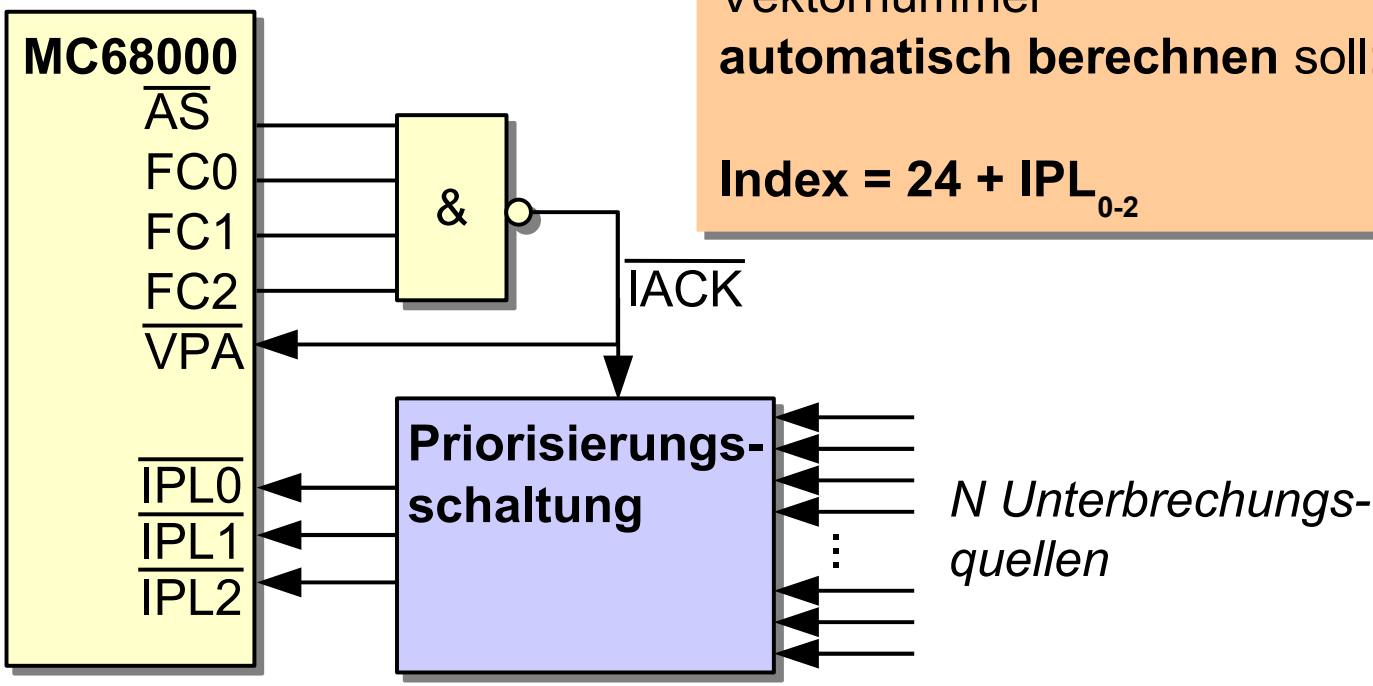
Vektortabelle des MC68000

Index	Adresse	Bedeutung
0	0x000	Reset: Supervisor-Stapelzeiger
1	0x004	Reset: PC
2	0x008	Busfehler
3	0x00c	Adressfehler
4	0x010	Illegaler Befehl
5	0x014	Division durch Null
...		
24	0x060	unechte Unterbrechung
25	0x064	**autovektorielle Unterbrechung, Ebene 1**
26	0x068	**autovektorielle Unterbrechung, Ebene 2**
...		
30	0x078	**autovektorielle Unterbrechung, Ebene 6**
31	0x07c	**autovektorielle Unterbrechung, Ebene 7 (NMI)**
32-47	0x080	TRAP-Befehlsvektoren
48-63	0x0c0	reserviert
64-255	0x100	**Anwender-Unterbrechungsvektoren**

Autovektorielle Unterbrechungen

über **VPA** signalisiert die externe Schaltung, dass die CPU die Vektornummer **automatisch berechnen** soll:

Index = 24 + IPL_{0-2}

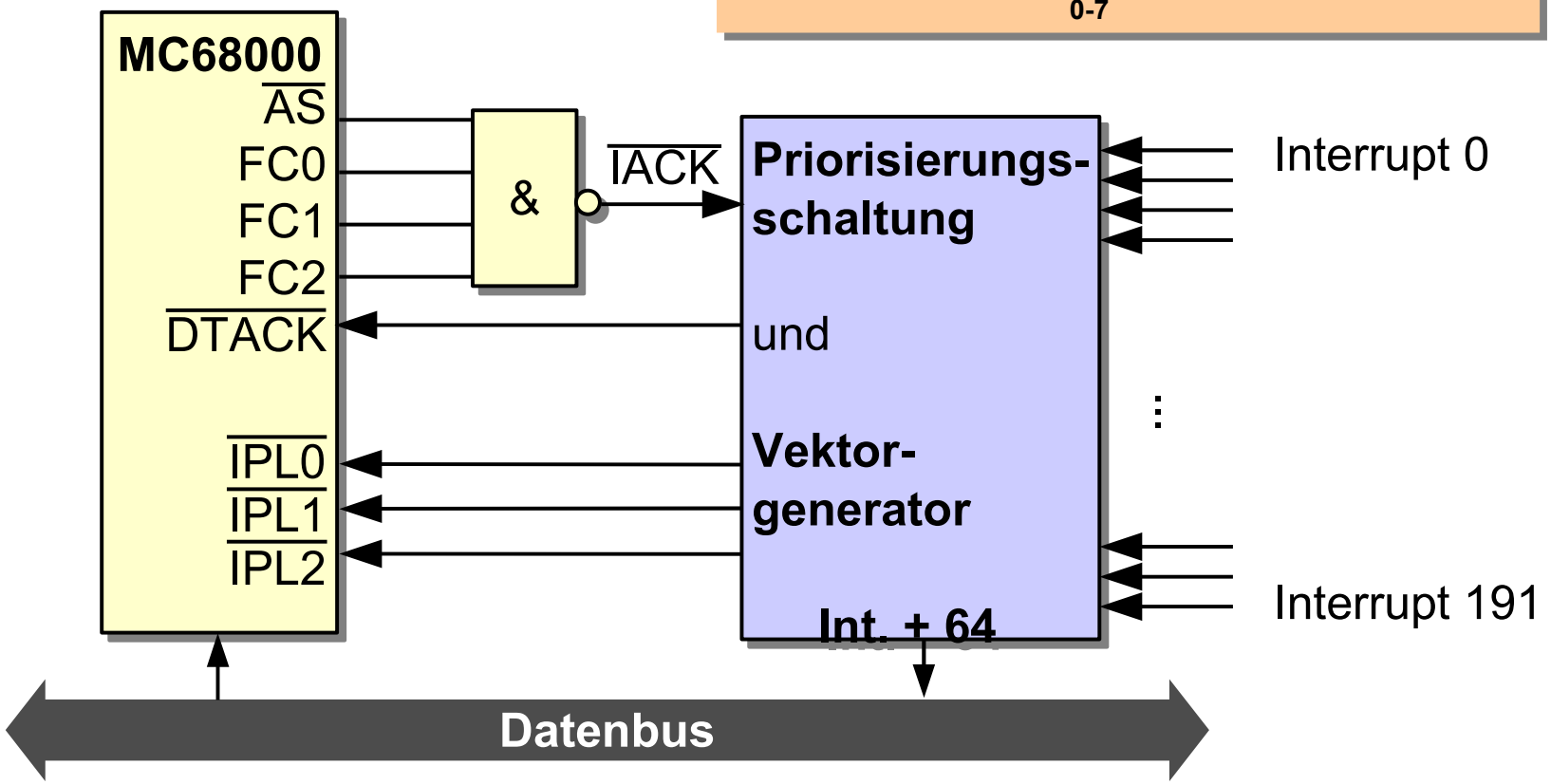


Problem: Es stehen nur 6 Vektoren für Geräte bereit.
Bei mehr Geräten ist „*sharing*“ nicht zu vermeiden.

Nicht-autovektorielle Unterbrechungen

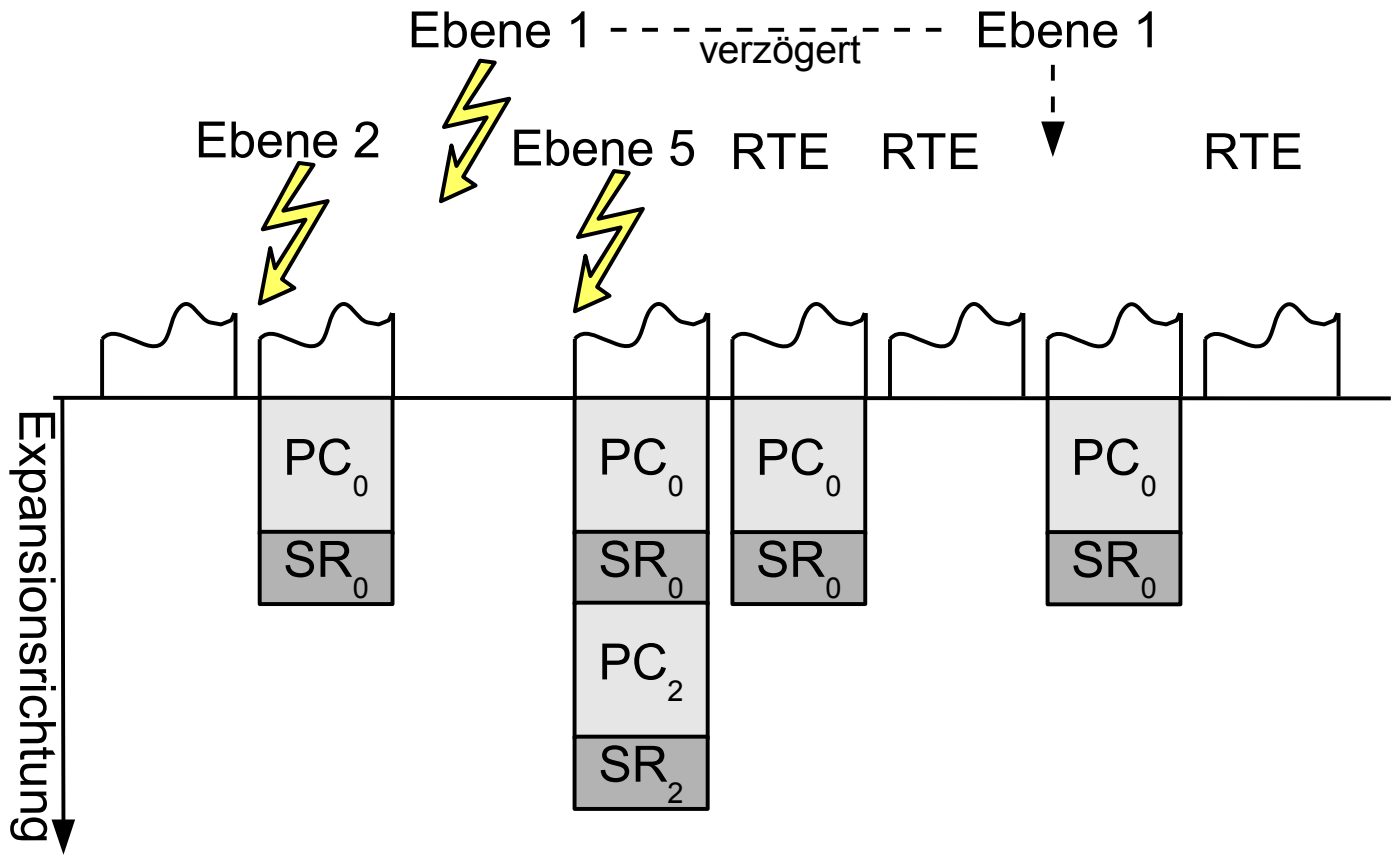
über **DTACK** signalisiert die externe Schaltung, dass die CPU die Vektornummer
über den Datenbus lesen soll.

Index = 64 + D_{0-7}



Zustandssicherung beim MC68000

- der vorherige SR Inhalt und der PC werden bei einer Unterbrechung auf dem Supervisor-Stapel gesichert
- der RTE Befehl macht den Vorgang rückgängig



MC68000 - Zusammenfassung

- 6 Prioritätsebenen für Hardware-Unterbrechungen + NMI
  - Unterbrechungsebene 1-6, NMI Ebene 7
  - „Maskierung“ über I_{0-2} im Statusregister möglich
- nur Unterbrechungen höherer Priorität und der NMI können eine laufende Behandlung unterbrechen
  - Statusregister wird automatisch angepasst
- automatische Zustandssicherung auf dem *Supervisor*-Stapel, geschachtelte Behandlung möglich.
- die Vektornummernerzeugung erfolgt entweder ...
  - autovektoriell: Index = Priorität + 24
  - nicht-autovektoriell (durch externe Hardware): Index = 64 ... 255
- keine Multiprozessorunterstützung

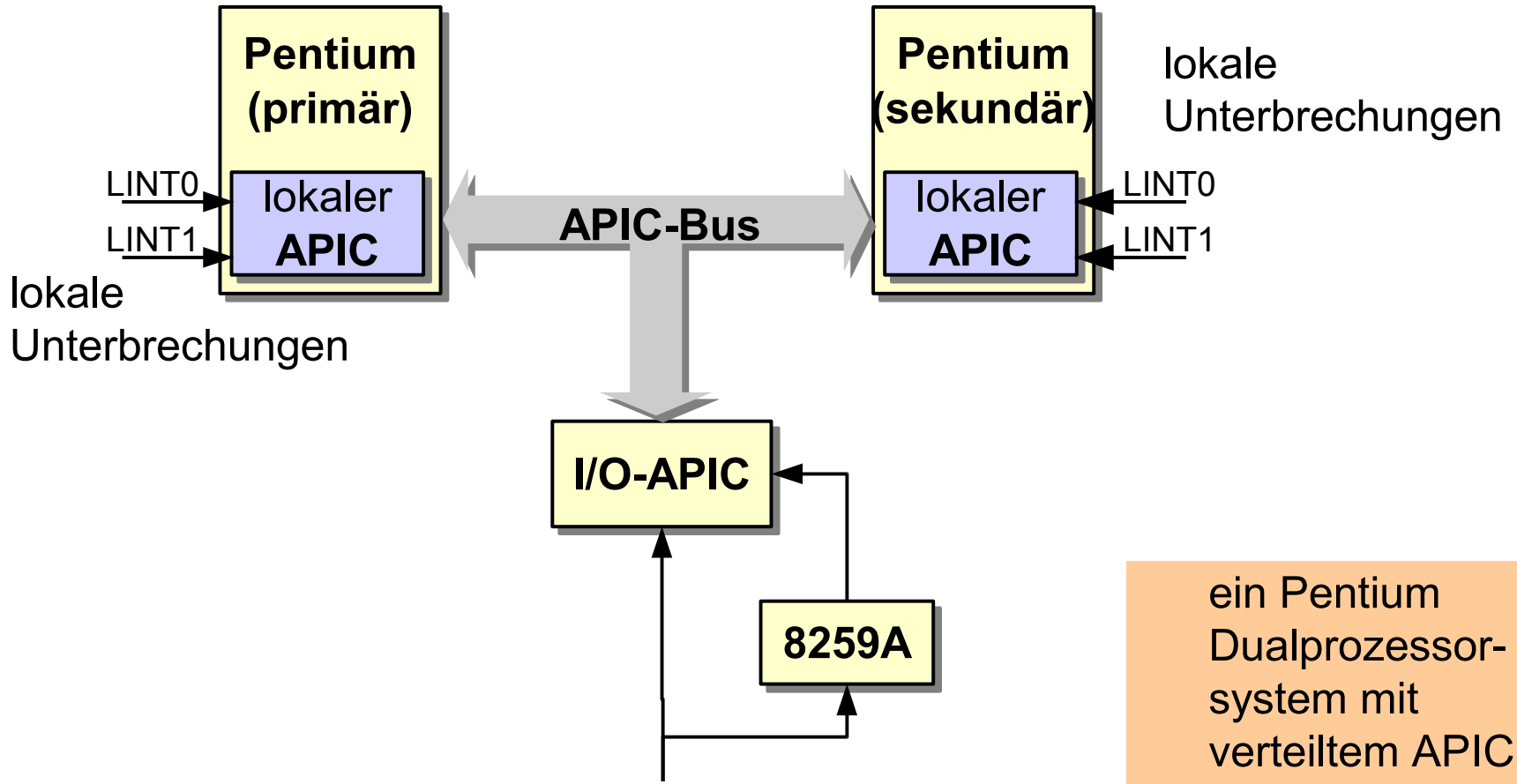
Unterbrechungen bei x86 CPUs

Unterbrechungen bei x86 CPUs

- bis einschließlich i486 hatten x86 CPUs nur einen IRQ und einen NMI Eingang
- externe Hardware sorgte für die Priorisierung und Vektornummerngenerierung
  - durch einen Chip namens **PIC 8259A**
    - 8 Interrupt-Eingänge
    - 15 Eingänge bei Kaskadierung von zwei PICs
    - keine Multiprozessorunterstützung
- heutige x86 CPUs enthalten den weit leistungsfähigeren **„Advanced Programmable Interrupt Controller“ (APIC)**
  - notwendig für **Multiprozessorsysteme**
  - inzwischen aber auch in allen Einprozessorsystemen aktiv
    - natürlich gibt es den PIC 8259A noch immer 🙂

Die APIC Architektur

- ein APIC *Interrupt*-System besteht aus lokalen APICs auf jeder CPU und einem I/O APIC



ein Pentium Dualprozessor-system mit verteiltem APIC Interrupt-System

Der I/O APIC

- heute typischerweise in der *Southbridge* von PC Chipsätzen integriert
- normalerweise 24 *Interrupt*-Eingänge
  - zyklische Abfrage (Round-Robin Priorisierung)
- für jeden Eingang gibt es einen 64 Bit Eintrag in der ***Interrupt Redirection Table***
  - beschreibt das Unterbrechungssignal
  - dient der Generierung der APIC-Bus Nachricht

Der I/O APIC

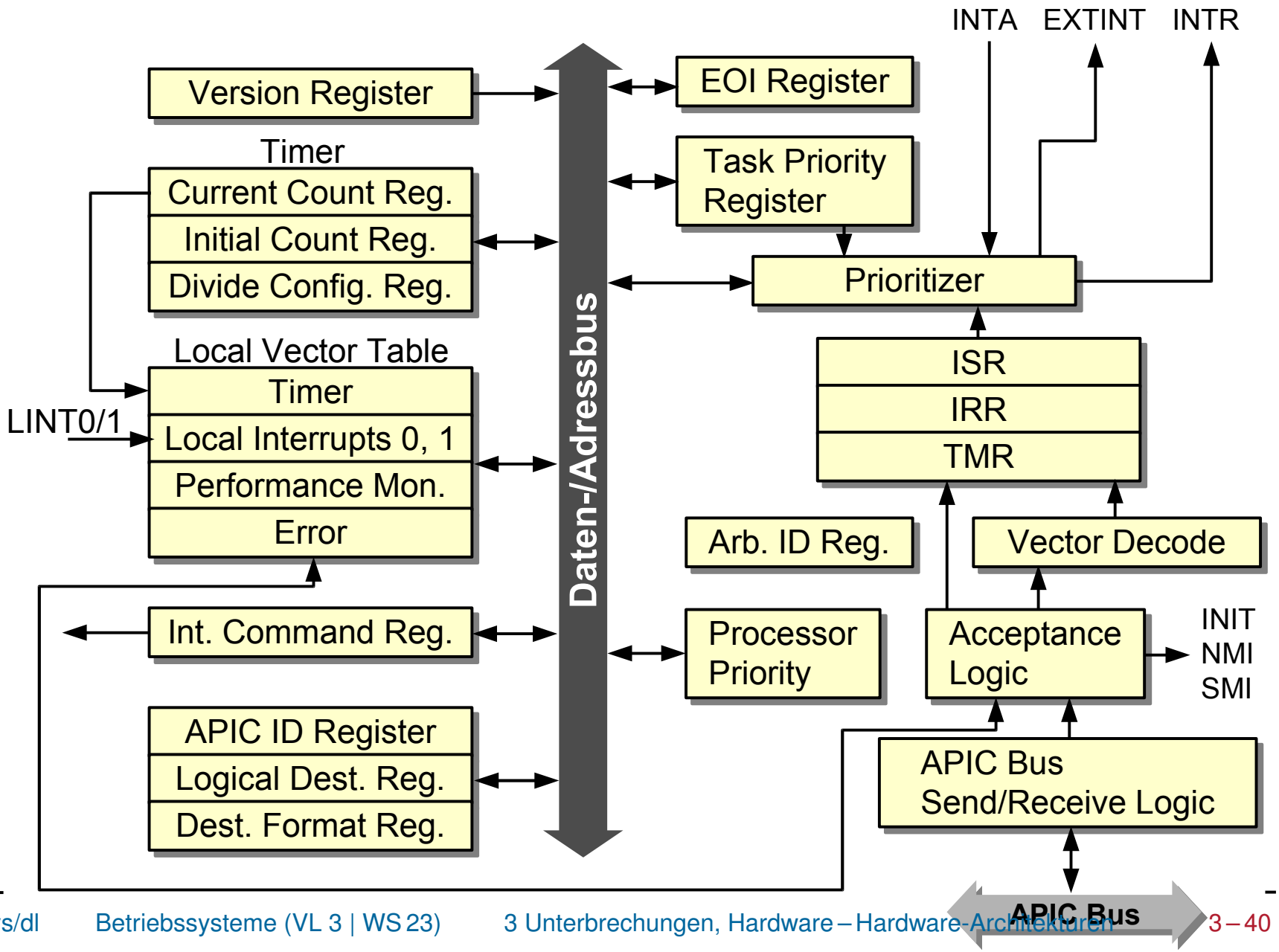
Aufbau (Bits) eines Eintrags in der *Interrupt Redirection Table*

63:56	**Destination Field** je nach Bit 11:	– R/W. 8 Bit Zieladresse. APIC ID der CPU (*Physical Mode*) oder CPU Gruppe (*Logical Mode*)
55:17	<reserviert>	
16	**Interrupt-Mask**	– R/W. Unterbrechungssperre.
15	**Trigger Mode**	– R/W. *Edge*- oder *Level-Triggered*
14	**Remote IRR**	– RO. Art der erhaltenen Bestätigung
13	**Interrupt Pin Polarity**	– R/W. Signalpolarität
12	**Delivery Status**	– RO. Interrupt-Nachricht unterwegs?
11	**Destination Mode**	– R/W. *Logical Mode* oder *Physical Mode*
10:8	**Delivery Mode**	– R/W. Wirkung bei Ziel-APIC
	000 – *Fixed*:	Signal an alle Zielprozessoren ausliefern
	001 – *Lowest Priority*:	Liefern an CPU mit aktuell niedrigster Prio.
	010 – SMI:	*System Management Interrupt*
	100 – NMI:	*Non-Maskable Interrupt*
	101 – INIT:	Ziel-CPUs initialisieren (Reset)
	111 – ExtINT:	Antwort an PIC 8259A
7:0	**Interrupt Vector**	– R/W. **8 Bit Vektornummer (16 – 254)**

Local APICs

- empfangen Unterbrechungsanforderungen vom APIC Bus
- führen die Auswahl und Priorisierung durch
- können zwei lokale Unterbrechungen direkt verarbeiten
- enthalten weitere Funktionseinheiten
  - Eingebauten *Timer*, *Performance Counter*
  - *Command*-Register
    - um selber APIC-Nachrichten zu verschicken
    - insbesondere Inter-Prozessor-Interrupt (IPI)
- programmierbar über 32 Bit Register ab 0xfee00000
  - memory mapped (ohne externe Buszyklen)
  - jede CPU programmiert „ihren“ *Local APIC*

Local APICs - Register



APIC Architektur - Zusammenfassung

- flexible Verteilung an CPUs im x86 Multiprozessorsystem
  - fest, Gruppen, an die CPU mit der geringsten Priorität
  - Liegen mehrere IRQs an, so wird nach Vektornummer priorisiert
- Vektornummer 16-254 können frei zugeordnet werden
  - sollte (an sich) reichen, um „sharing“ zu vermeiden
- *Local* APIC erwartet explizites EOI
  - dafür muss die Software sorgen
- Mit APIC unterstützt x86 prinzipiell auch Prioritätsebenen
  - Systemsoftware muss jedoch entsprechend agieren (Unterbrechungen freigeben, evtl. *Task-Priority-Register* verwenden)

Agenda

Einordnung
Grundlagen
Hardware-Architekturen
Zusammenfassung

Zusammenfassung und Ausblick

- Unterbrechungsbehandlungshardware befasst sich mit ...
  - Priorisierung
  - Zuordnung und Ausführung einer Behandlungsroutine
  - Zustandssicherung und geschachtelter Ausführung
- moderne Unterbrechungsbehandlungshardware kann ...
  - Unterbrechungsvektoren frei zuordnen
  - „*sharing*“ von Unterbrechungsvektoren vermeiden
  - Unterbrechungen im Multiprozessorsystem flexibel zuordnen
- das Betriebssystem muss ...
  - Probleme wie „*spurious interrupts*“ und „*interrupt storms*“ einkalkulieren.
  - das eingetretene Ereignis aus der Behandlungsroutine an die höheren Ebenen und letztendlich zum Anwendungsprozess weiterleiten.

Betriebssysteme (BS)

VL 4 – Unterbrechungen, Software

Volkmar Sieh / **Daniel Lohmann**

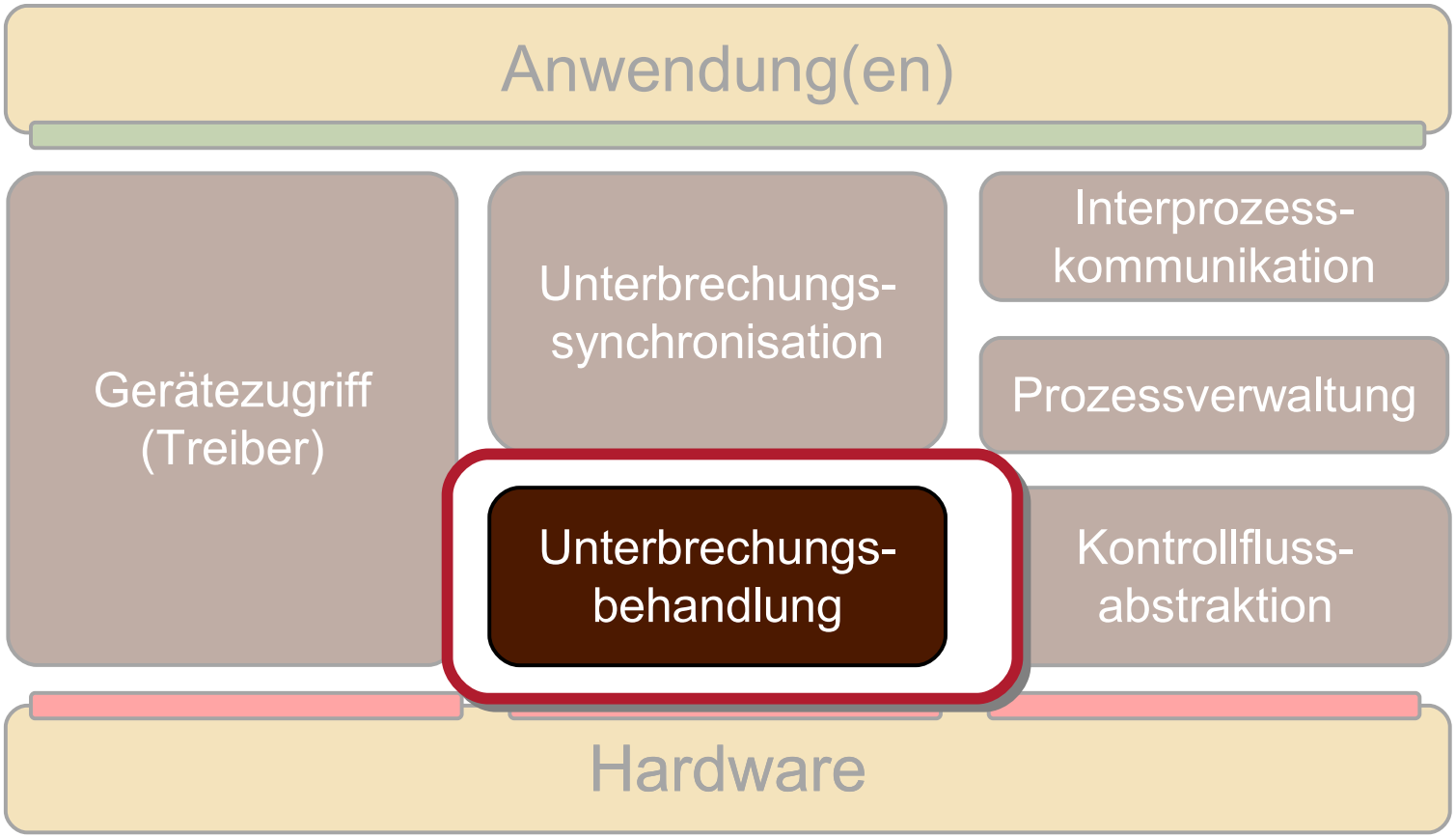
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 15. November 2023

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

Einordnung
Begriffe und Grundannahmen
Zustandssicherung
Zustandsänderungen
Zusammenfassung

Agenda

Einordnung

Begriffe und Grundannahmen

- Interrupt, Exception, Trap
- Grundannahmen

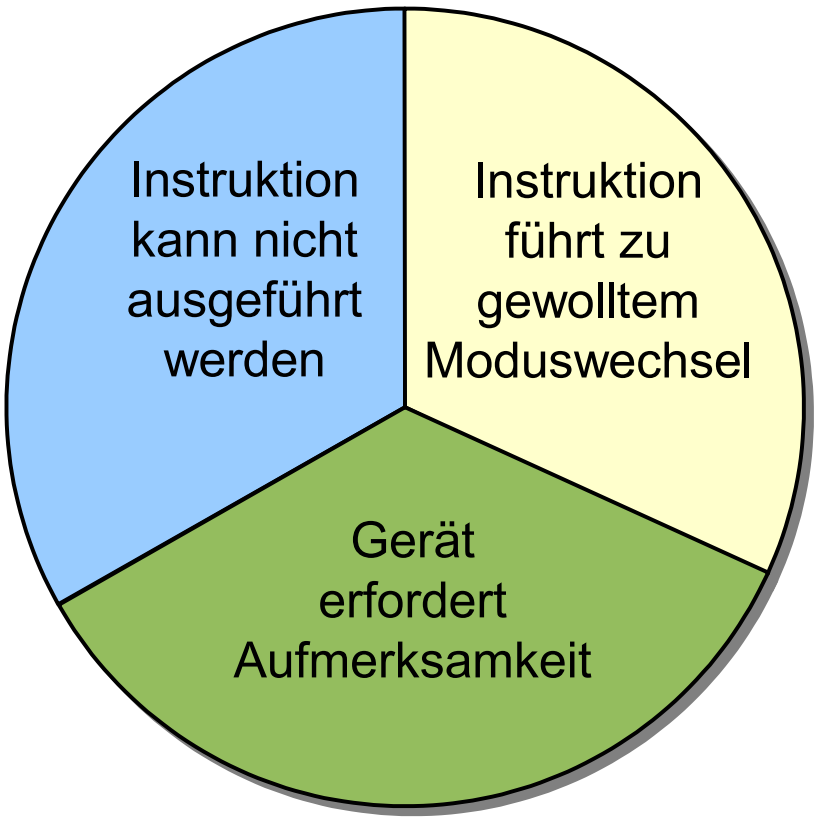
Zustandssicherung

Zustandsänderungen

Zusammenfassung

Begriffe

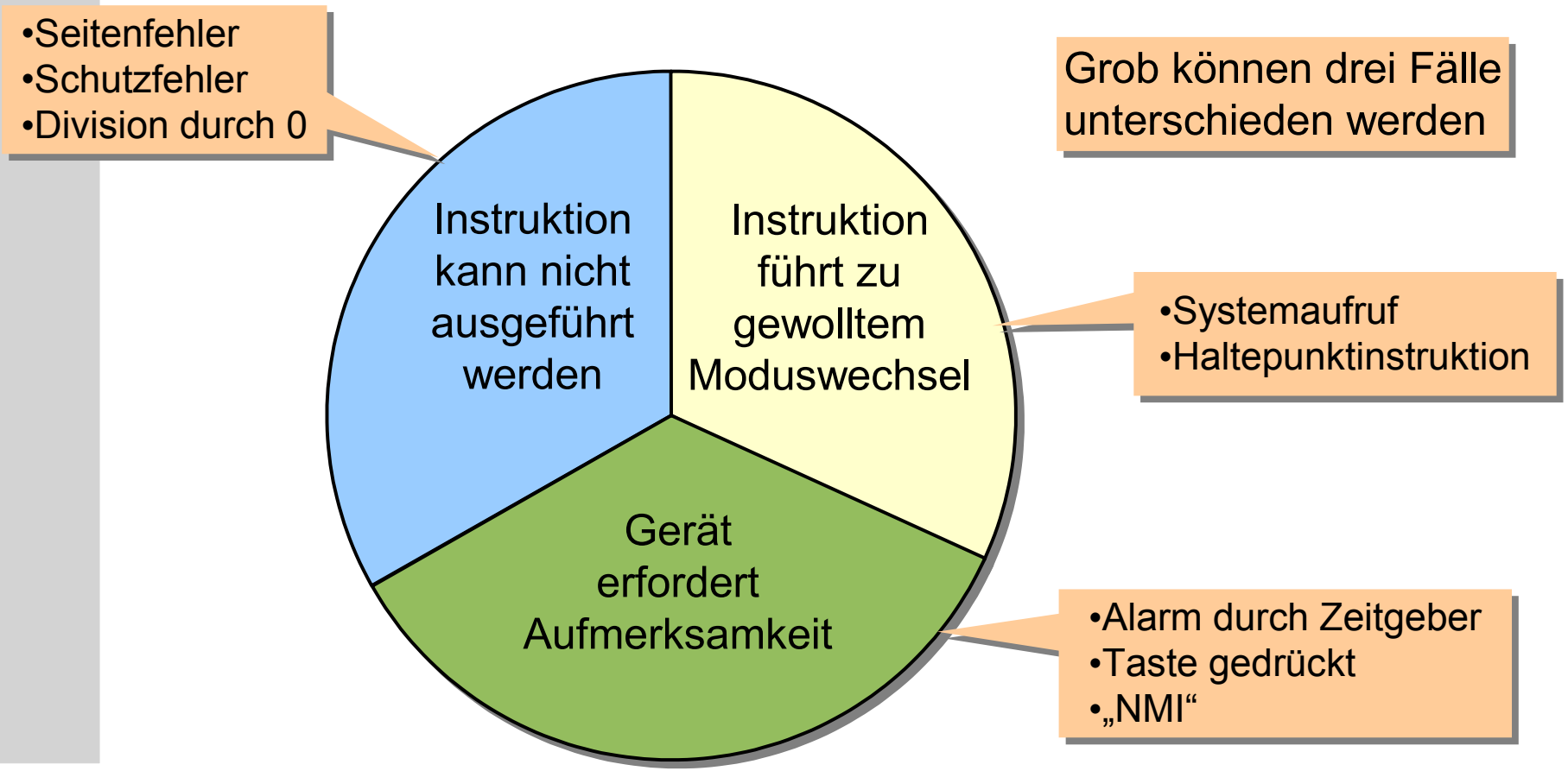
- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



Grob können drei Fälle unterschieden werden

Begriffe

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene

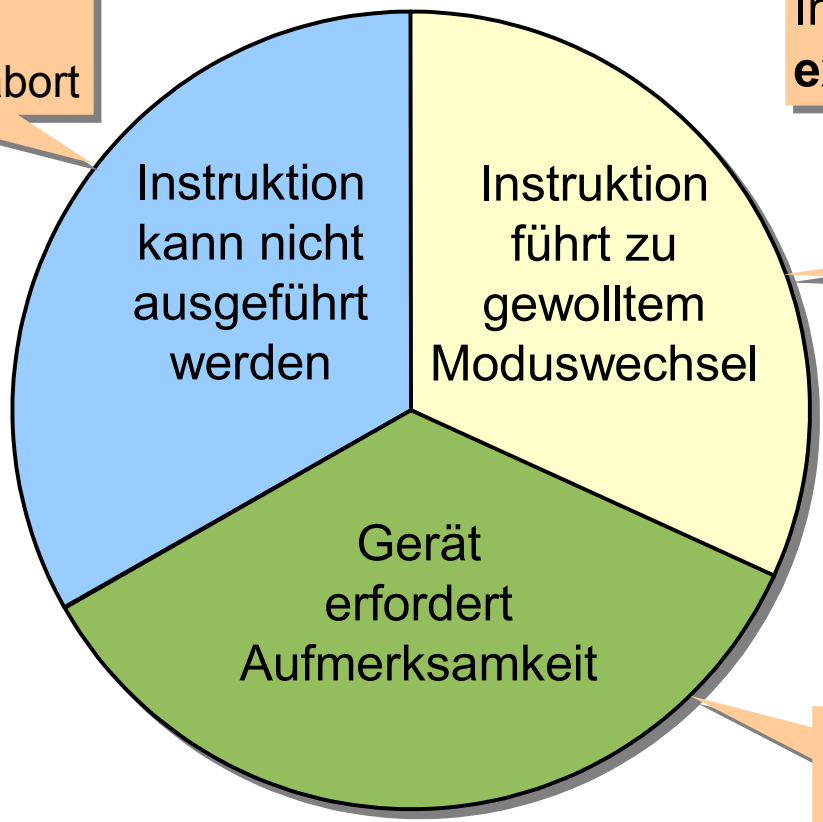


Begriffe: Intel IA-32

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene

(software-generated) **exceptions**
• fault, trap oder abort

Intel **IA-32-Architektur** (x86): **exceptions** und **interrupts**

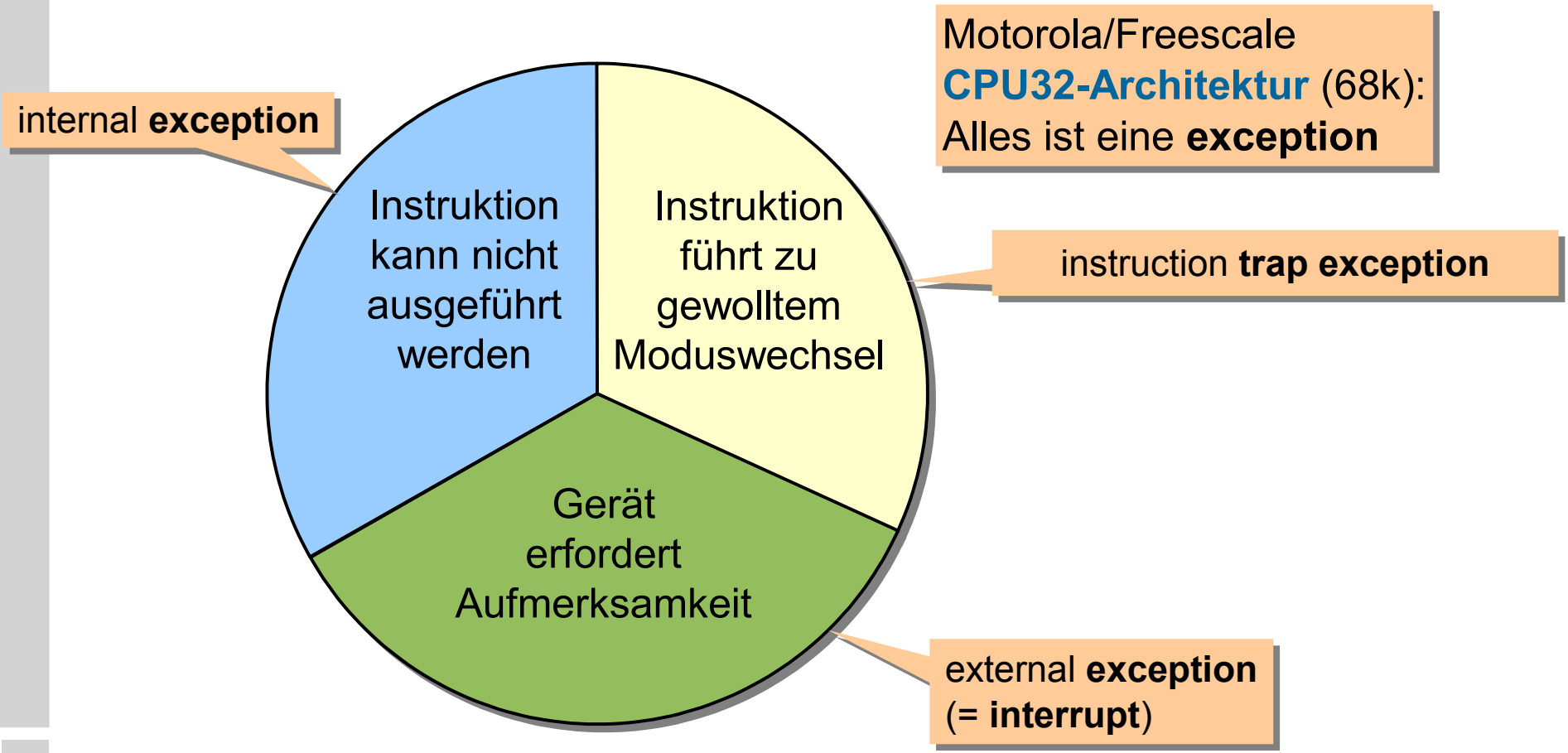


software(-generated) **interrupts**

external (hardware-generated) **interrupts**

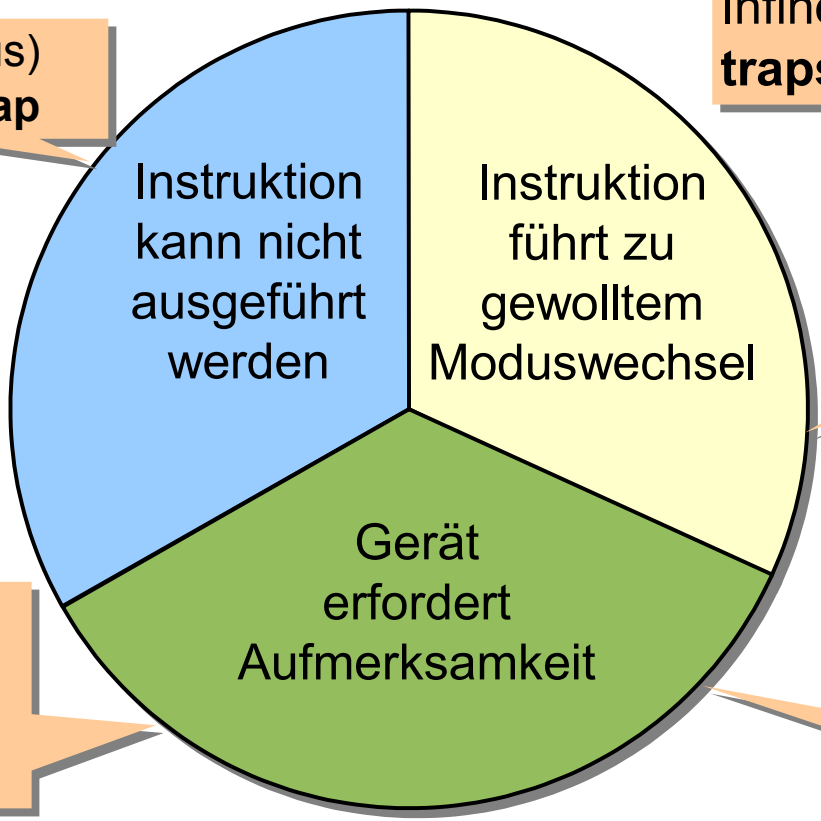
Begriffe: Motorola/Freescale CPU32

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



Begriffe: Infineon TC1

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



Infineon **TC1-Architektur** (TriCore): **traps** und **interrupts**

(synchronous) hardware **trap**

(synchronous) software **trap**

(asynchronous) hardware **trap**
- „NMI“
- DMA-Fehler

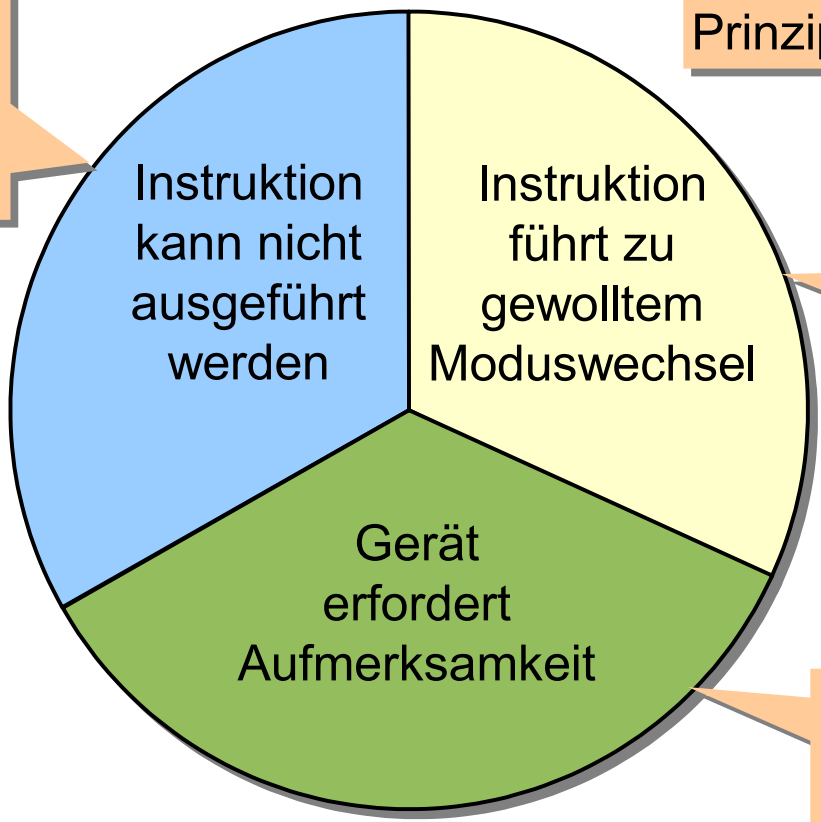
interrupt

Begriffe: Literatur (Stallings)

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene

Trap
Behandlung eines Fehlers oder eines Ausnahme-zustandes

William Stallings: „Betriebssysteme: Prinzipien und Umsetzung“



Supervisor-Aufruf
Aufruf einer Betriebssystemfunktion

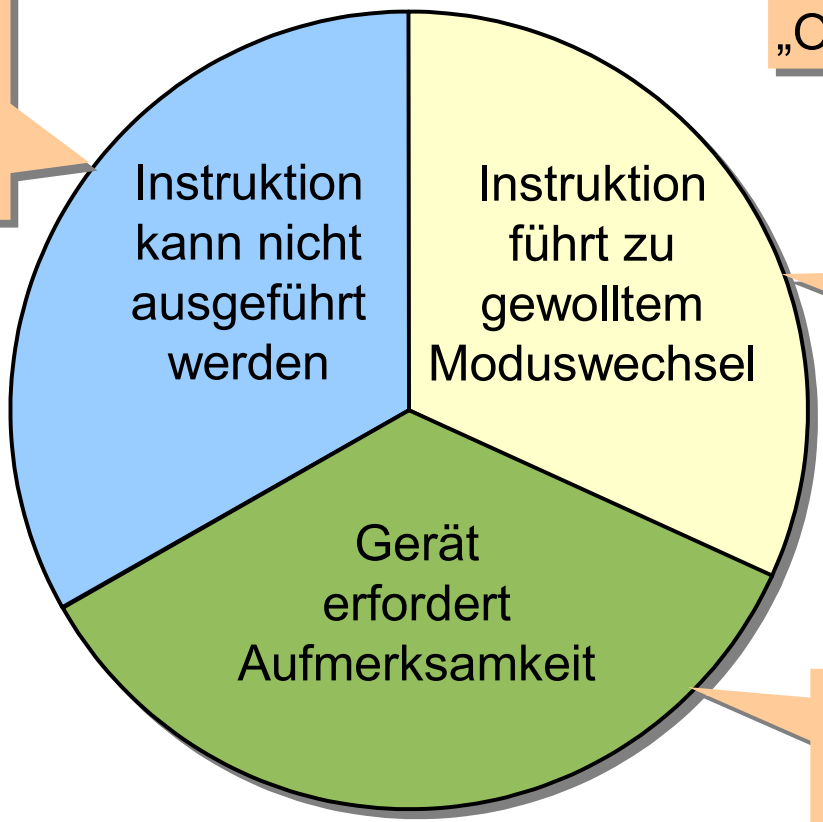
Interrupt
Reaktion auf ein externes asynchrones Ereignis

Begriffe: Literatur (Silberschatz)

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene

exception
Behandlung eines Fehlers oder eines Ausnahme-zustandes

Silberschatz *et al* :
„Operating System Concepts“



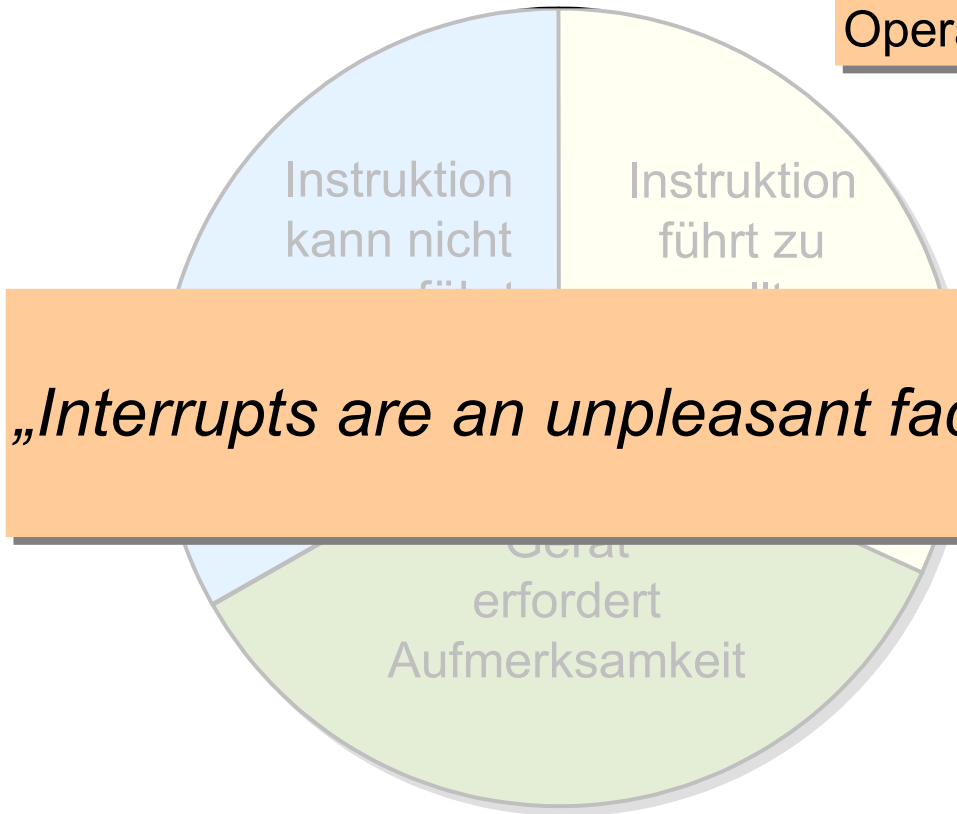
software interrupt / trap
Aufruf einer Betriebssystemfunktion

interrupt
Reaktion auf ein externes asynchrones Ereignis

Begriffe: Literatur (Tanenbaum)

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene

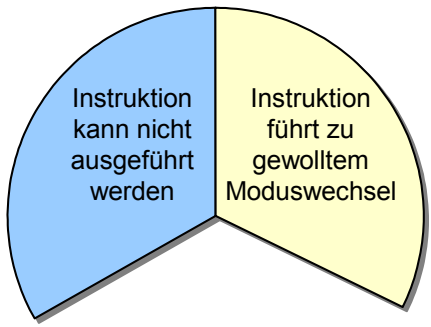
Andrew S. Tanenbaum: „Modern Operating Systems“ (ältere Ausgabe)



„Interrupts are an unpleasant fact of life“

Begriffe: Unser Verständnis in BS

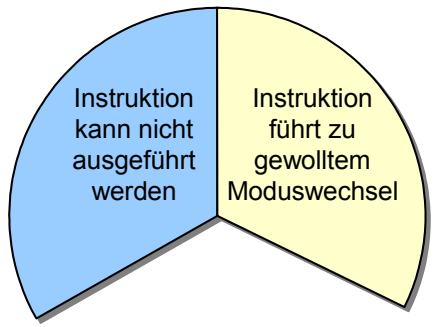
- „*Trap*“
  - durch **Instruktion** ausgelöst
    - auch die „trap“ oder „int“ Instruktion für Systemaufrufe
    - nicht definiertes Ergebnis (z.B. Division durch 0)
    - Hardware-Problem (z.B. Busfehler)
    - Betriebssystem muss eingreifen (z.B. Seitenfehler)
    - ungültige Instruktion (z.B. bei Programmfehler)
  - Eigenschaften
    - oft vorhersagbar, oft reproduzierbar
    - Wiederaufnahme **oder Abbruch** der auslösenden Aktivität



- **„Unterbrechung“** (engl. *Interrupt*):
  - durch **Hardware** ausgelöst
    - Hardware verlangt die Aufmerksamkeit der Software (Zeitgeber, Tastatursteuereinheit, Festplattensteuereinheit, ...)
  - Eigenschaften:
    - nicht vorhersagbar, nicht reproduzierbar
    - in der Regel Wiederaufnahme der unterbrochenen Aktivität

Begriffe: Unser Verständnis in BS

- „*Trap*“
  - durch **Instruktion** ausgelöst
    - auch die „trap“ oder „int“ Instruktion für Systemaufrufe
    - nicht definiertes Ergebnis (z.B. Division durch 0)
    - Hardware-Problem (z.B. Busfehler)
    - Betriebssystem muss eingreifen (z.B. Seitenfehler)
    - ungültige Instruktion (z.B. bei Programmfehler)
  - Eigenschaften
    - oft vorhersagbar, oft reproduzierbar
    - Wiederaufnahme **oder Abbruch** der auslösenden Aktivität



- **„Unterbrechung“** (engl. *Interrupt*):
  - durch **Hardware** ausgelöst
    - Hardware verlangt die Aufmerksamkeit der Software (Zeitgeber, Tastatursteuereinheit, Festplattensteuereinheit, ...)
  - Eigenschaften:
    - nicht vorhersagbar, nicht reproduzierbar
    - in der Regel Wiederaufnahme der unterbrochenen Aktivität

Grundannahmen

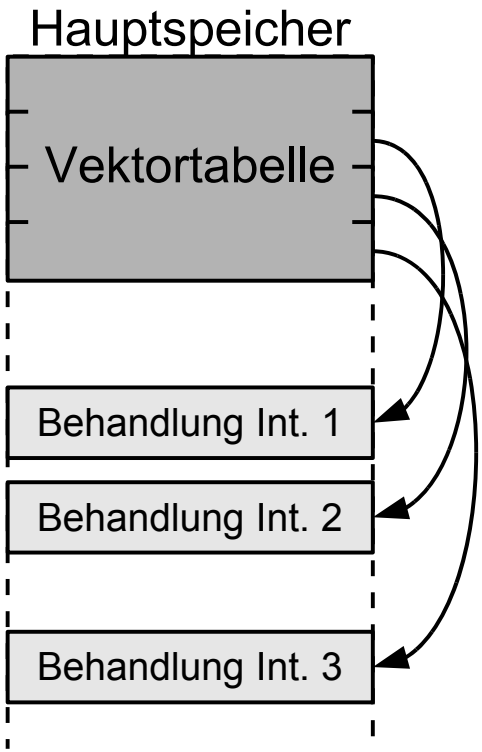
Wir betrachten die Behandlung von Unterbrechungen unter den folgenden Grundannahmen

1. Die CPU startet die Behandlungsroutine automatisch.
2. Die Unterbrechnungsbehandlung erfolgt im Systemmodus.
3. Das unterbrochene Programm kann fortgesetzt werden.
4. Die Maschineninstruktionen verhalten sich atomar.
5. Die Unterbrechungsbehandlung kann unterdrückt werden.

Grundannahmen: Behandlungsroutine

1. Die CPU startet die Behandlungsroutine automatisch.

- erfordert die Zuordnung einer Behandlungsroutine
- Ermittlung der Unterbrechungsursache nötig



Varianten:
- Register enthält Startadresse der Vektortabelle
- Tabelleneinträge enthalten Code
- Programmierbarer „Event Controller“ behandelt die Unterbrechung in Hardware
- Tabelle enthält Deskriptoren
- Behandlungsroutine hat eigenen Prozesskontext

Grundannahmen: Systemmodus

2. Die Unterbrechungsbehandlung erfolgt im Systemmodus.

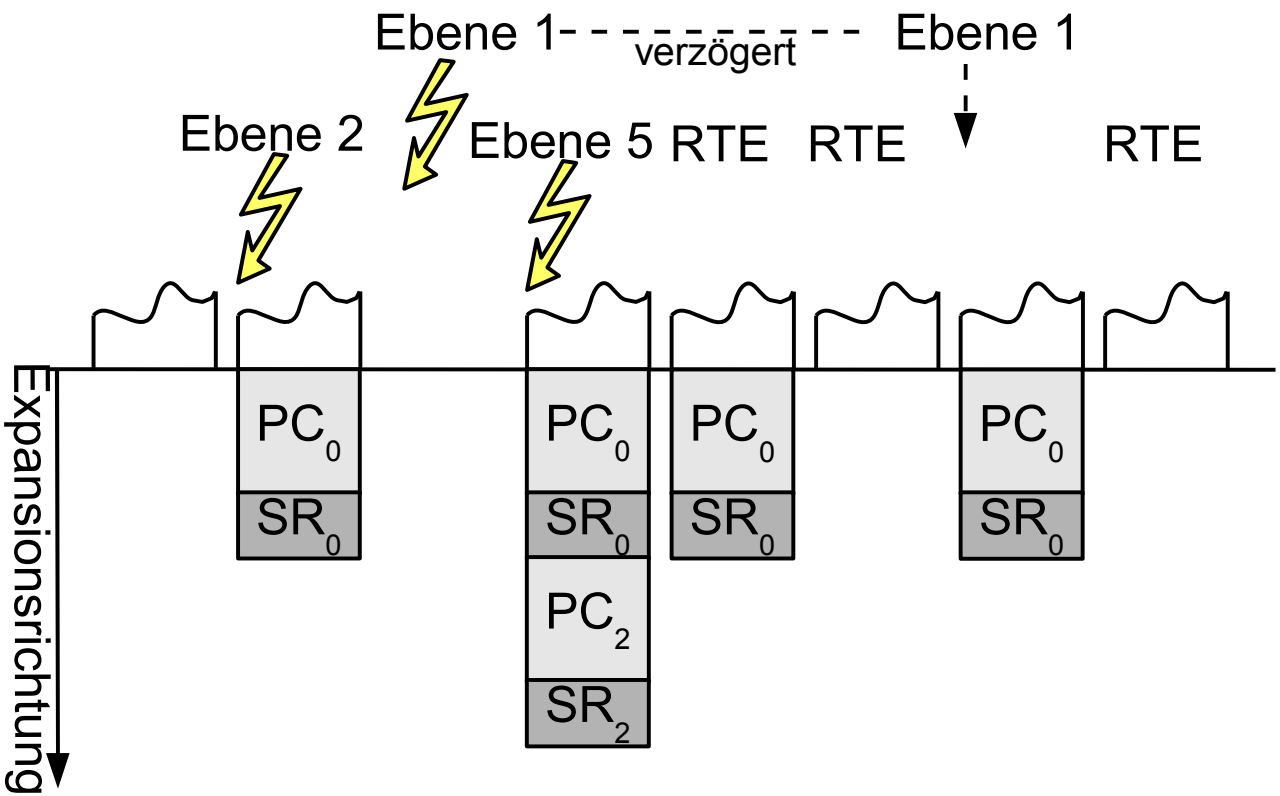
- Unterbrechungen sind der einzige Mechanismus, um nicht-kooperativen Anwendungen die CPU zu entziehen
- nur das BS darf uneingeschränkt auf Geräte zugreifen
- die CPU schaltet daher vor der Unterbrechungs-behandlung in den privilegierten Systemmodus

Varianten:
- bei **16-Bit-CPUs** ist eine Aufteilung in Benutzer-/Systemmodus eher die Ausnahme
- bei **8-Bit-CPUs** (oder kleiner) gibt es diese Aufteilung nicht

Grundannahmen: Kontextsicherung

3. Das unterbrochene Programm kann fortgesetzt werden.

- notwendiger Zustand wird automatisch gesichert
- ggf. auch geschachtelt, erfordert Stapel



Varianten:
- weitere Informationen über die Ursache auf dem Stapel
- keine Prioritäten
- spezieller „Interrupt-Stack“
- Kontextsicherung in Registern

Grundannahmen: Atomares Verhalten

4. Die Maschineninstruktionen verhalten sich atomar.

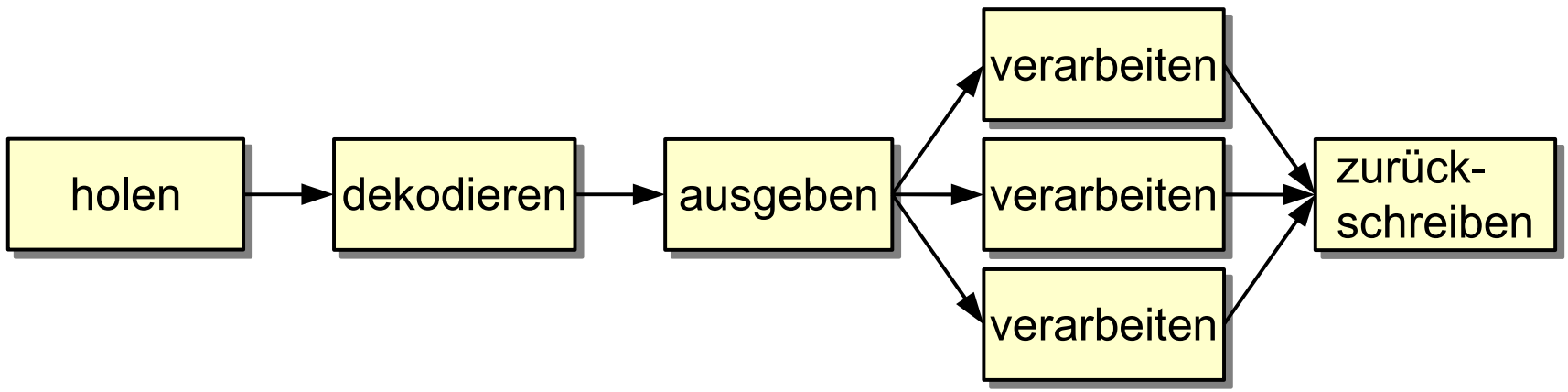
- definierter CPU-Zustand zu Beginn der Behandlungsroutine
- Wiederherstellbarkeit des Zustands

- trivial bei CPUs mit klassischem von-Neumann-Zyklus
- nicht-trivial bei modernen CPUs:
  - Fließbandverarbeitung: Befehle müssen annuliert werden
  - Superskalare CPUs: zusätzlich Befehlsreihenfolge merken

Grundannahmen: Atomares Verhalten

4. Die Maschineninstruktionen verhalten sich atomar.

Befehlsverarbeitung bei superskalaren Prozessoren:
(stark vereinfacht!)



Im Idealfall werden alle Stufen immer benutzt, d.h. mehrere Befehle werden parallel ausgeführt. Wann soll geprüft werden, ob eine Unterbrechungsanforderung anliegt?

Grundannahmen: Atomares Verhalten

4. Die Maschineninstruktionen verhalten sich atomar.

Trotz der Schwierigkeiten liefern die meisten CPUs „**präzise Unterbrechungen**“:

- *„All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.“*
- *„All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.“*
- *„If the interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.“*

J. E. Smith and A. R. Pleszkun,
„Implementing Precise Interrupts in Pipelined Processors“,
IEEE Transactions on Computers, Vol. 37, No. 5, 1988

Grundannahmen: Unterdrückung

5. Die Unterbrechungsbehandlung kann unterdrückt werden.

- Beispiele:
  - Motorola 680x0: entsprechend der Priorität
  - Intel x86: global mit `sti`, `cli`
  - *Interrupt Controller*: jede Quelle einzeln
- automatische Unterdrückung erfolgt auch durch die CPU vor Betreten der Behandlungsroutine

Grundannahmen: Unterdrückung

5. Die Unterbrechungsbehandlung kann unterdrückt werden.

- automatische Unterdrückung erfolgt auch durch die CPU vor Betreten der Behandlungsroutine
  - Unterbrechungen nicht vorhersagbar, theoretisch beliebig häufig
  - Stapelüberlauf könnte nicht ausgeschlossen werden
- unterdrückt wird (durch die Hardware) die Behandlung...
  - pauschal aller Unterbrechungen (sehr restriktiv)
  - Unterbrechungen niedriger oder gleicher Priorität (weniger restriktiv)
    - bestimmte Geräte werden bevorzugt
- bessere Modelle mit Hilfe von Software (z.B. in Linux):
  - Unterbrechungen, die bereits behandelt werden, werden unterdrückt
    - hohe Reaktivität ohne Bevorzugung einzelner Geräte

Agenda

Einordnung
Begriffe und Grundannahmen
Zustandssicherung
Zustandsänderungen
Zusammenfassung

Zustandssicherung

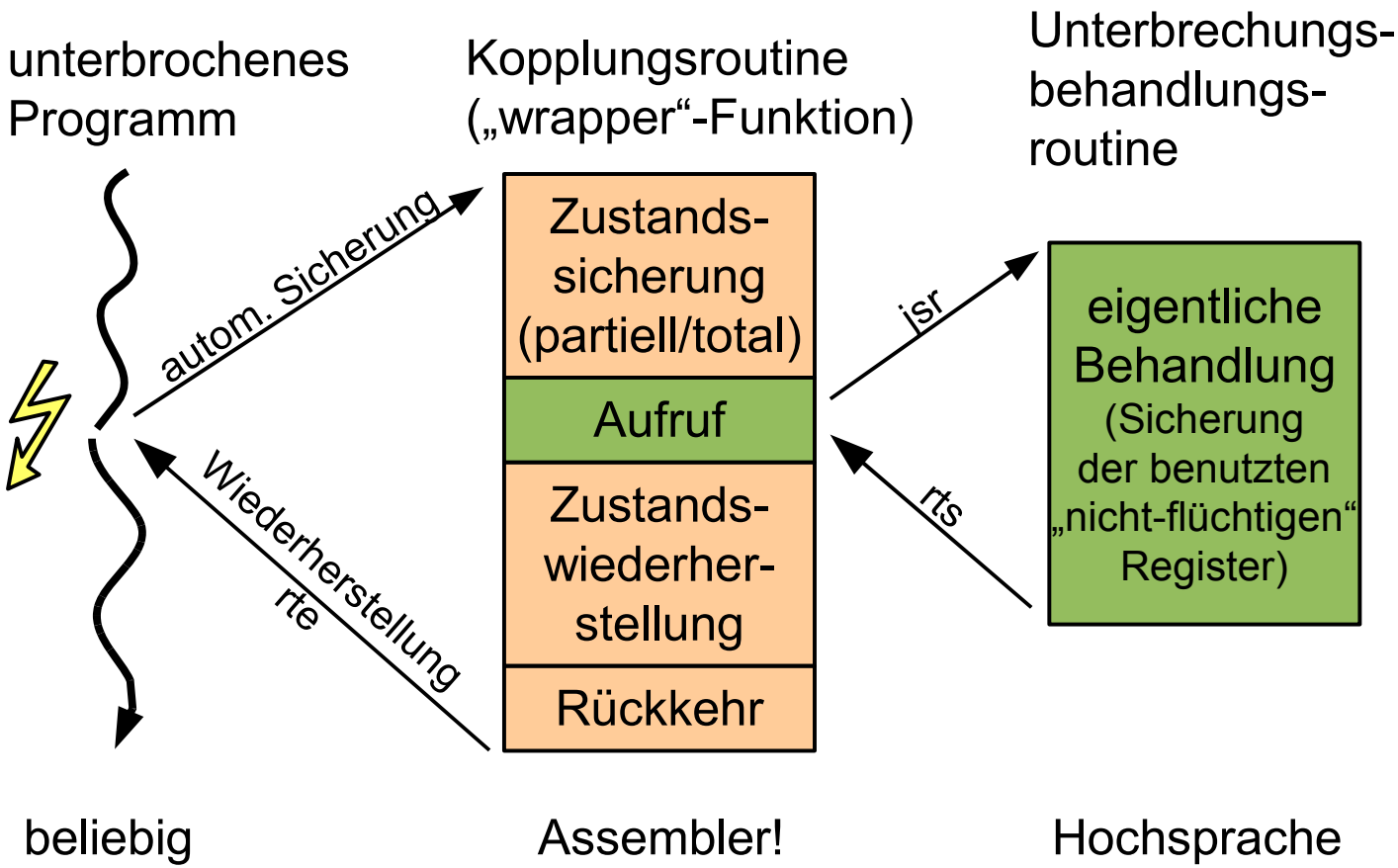
- der Zustand eines Rechners ist enorm groß
  - alle Prozessorregister
    - Instruktionszeiger, Stapelzeiger, Vielzweckregister, Statusregister, ...
  - der komplette Hauptspeicherinhalt, Caches
  - der Inhalt von E/A-Registern bzw. *Ports*, Festplatteninhalte, ...
- jeglicher benutzter Zustand, dessen asynchrone Änderung das unterbrochene Programm nicht erwartet, ...
  - darf während der Unterbrechungsbehandlung nicht modifiziert werden
  - muss gesichert und später wiederhergestellt werden
- die CPU sichert (je nach Typ) automatisch ...
  - minimal wenige Bytes (nur Instruktionszeiger und Statusregister)
  - alle Register

Zustandssicherungskonzepte

- **totale Sicherung**
  - die Behandlungsroutine sichert alle Register, die nicht automatisch gesichert wurden
  - Nachteil: eventuell wird zu viel gesichert
  - Vorteil: gesicherter Zustand leicht „zugreifbar“
- **partielle Sicherung**
  - die Behandlungsroutine sichert nur die Register, die im weiteren Verlauf geändert werden bzw. nicht gesichert und wieder hergestellt werden
  - machbar, wenn die eigentliche Behandlung in einer Hochsprache wie C oder C++ implementiert ist
  - Vorteile:
    - nur veränderter Zustand wird auch gesichert
    - evtl. weniger Instruktionen zum Sichern und Wiederherstellen nötig
  - Nachteil: gesicherter Zustand „verstreut“

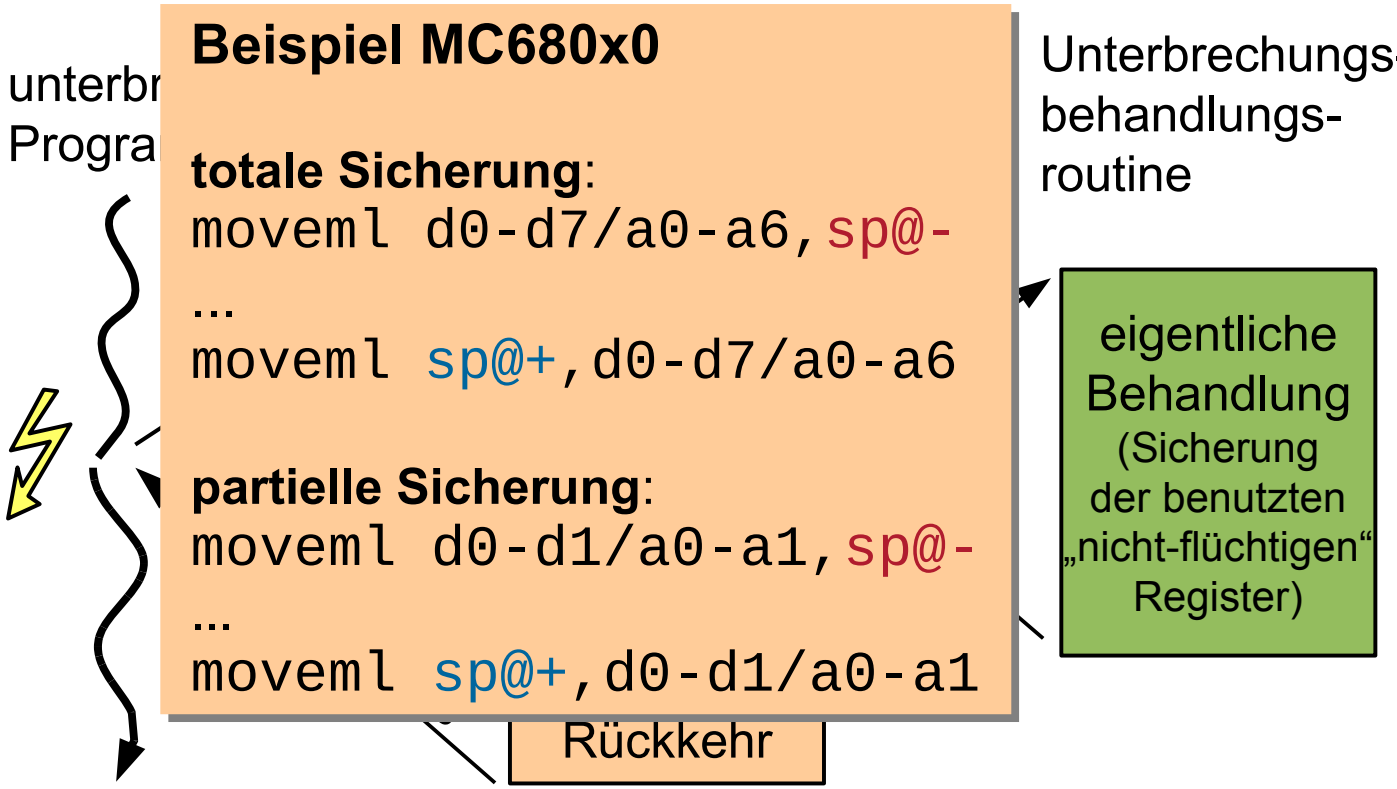
Übergang auf die Hochsprachenebene

- nicht-portabler Maschinencode sollte minimiert werden
- die eigentliche Unterbrechungsbehandlung erfolgt in einer Hochsprachenfunktion



Übergang auf die Hochsprachenebene

- nicht-portabler Maschinencode sollte minimiert werden
- die eigentliche Unterbrechungsbehandlung erfolgt in einer Hochsprachenfunktion



Sprache: beliebig Assembler! Hochsprache

Flüchtige und nicht-flüchtige Register

- eine Aufteilung, die **der (C/C++) Übersetzer** vornimmt
  - **nicht-flüchtig**
    - der Übersetzer garantiert, dass der Wert dieser Register über Funktionsaufrufe hinweg erhalten bleibt
    - ggf. in der aufgerufenen Funktion gesichert und wiederhergestellt
  - **flüchtig** (engl. *scratch registers*)
    - wenn die aufrufende Funktion den Wert auch nach dem Aufruf noch benötigt, muss das Register selbst (beim Aufrufer) gesichert werden
    - normalerweise für Zwischenergebnisse verwendet
- üblicherweise gibt es jedoch einen Standard
  - an den sich alle Übersetzer halten
  - Beispiel x86:
    - `eax`, `ecx`, `edx` und `eflags` gelten als flüchtig

Wiederherstellung

- die Kopplungsroutine muss alle gesicherten Registerinhalte am Ende wieder laden
  - ... und dann nicht mehr verändern!
- mit einer speziellen Instruktion (z.B. `rte` oder `iret`) wird der vorherige Zustand wiederhergestellt
  - Lesen des automatisch gesicherten Zustands von *Supervisor-Stack*
  - Setzen des gesicherten Arbeitsmodus (Benutzer-/Systemmodus) und Sprung an die gesicherte Adresse

Das BS kann den Zustand auch vor dem `rte`/`iret` ändern. Dies wird gerne ausgenutzt, um BS-Code im Benutzermodus auszuführen.

Agenda

Einordnung
Begriffe und Grundannahmen
Zustandssicherung
Zustandsänderungen
- Beispiele
- Problemanalyse
- Lösungsansätze

Zusammenfassung

Zustandsänderungen ...

- sind Sinn und Zweck der Unterbrechungsbehandlung
  - Gerätetreiber müssen über den Abschluss einer E/A Operation informiert werden
  - der Scheduler muss erfahren, dass eine Zeitscheibe abgelaufen ist
- müssen mit Vorsicht durchgeführt werden
  - Unterbrechungen können zu jeder Zeit auftreten
  - kritisch sind Daten/Datenstrukturen, die der normale Kontrollfluss und die Unterbrechungsbehandlung sich teilen

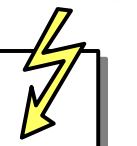
Beispiel 1: Systemzeit

- per Zeitgeberunterbrechung wird die globale Systemzeit inkrementiert
  - z.B. einmal pro Sekunde
- mit Hilfe einer Betriebssystemfunktion `time()` kann die Systemzeit abgefragt werden

```
/* globale Zeitvariable */
extern volatile time_t global_time;
```

```
/* Systemzeit abfragen */

time_t time () {
  return global_time;
}
```

```
/* Unterbrechungs- *
 * behandlung      */
void timerHandler () {
  global_time++;
}
```



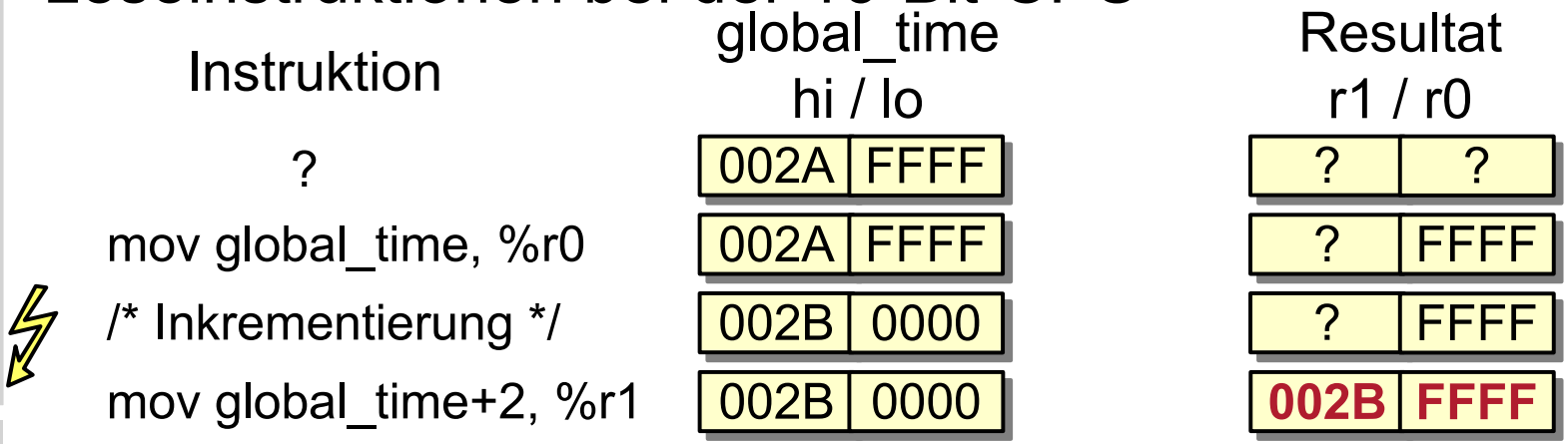
Beispiel 1: Systemzeit

- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global_time erfolgt nicht notwendigerweise atomar!

```
32-Bit-CPU:
mov global_time, %eax
```

```
16-Bit-CPU (little endian):
mov global_time, %r0; lo
mov global_time+2, %r1; hi
```

- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16-Bit-CPU



Beispiel 1: Systemzeit

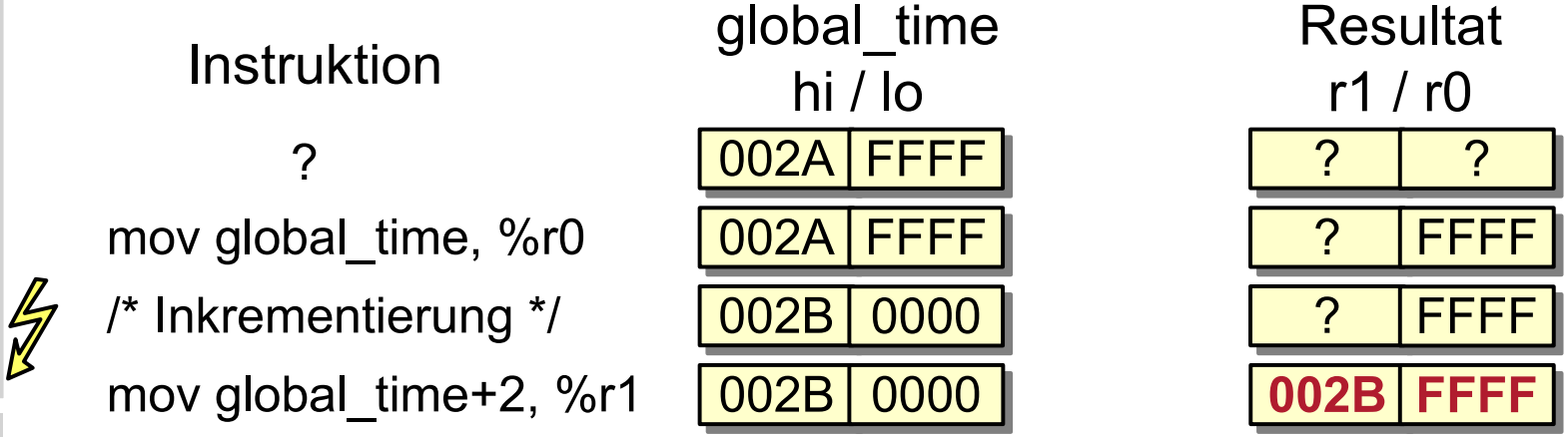
- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global_time erfolgt nicht notwendigerweise atomar!

```
32 Bit C                          le endian):
mov glob                           %r0; lo
                                  -2, %r1; hi
```

Problem:

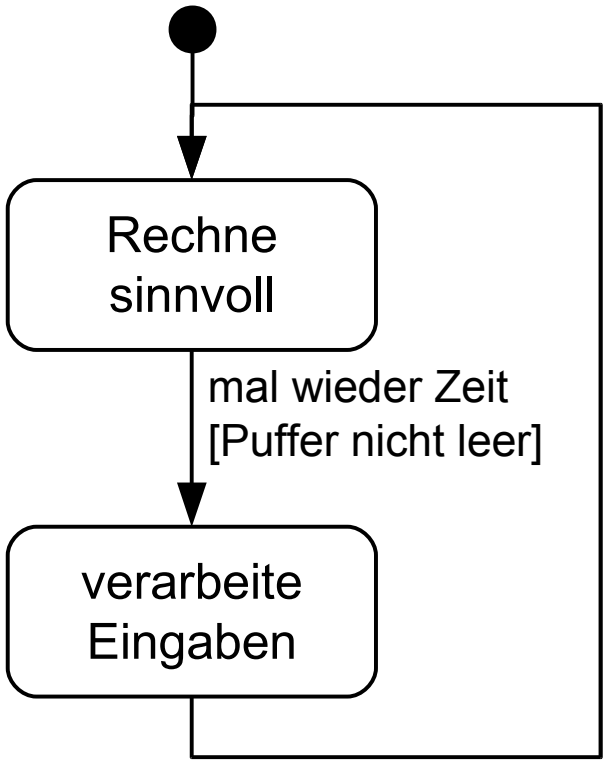
Alle 18,2 Stunden kann die Systemzeit (kurz) um etwa die gleiche Zeit vorgehen. Leider ist das Problem nicht verlässlich reproduzierbar.

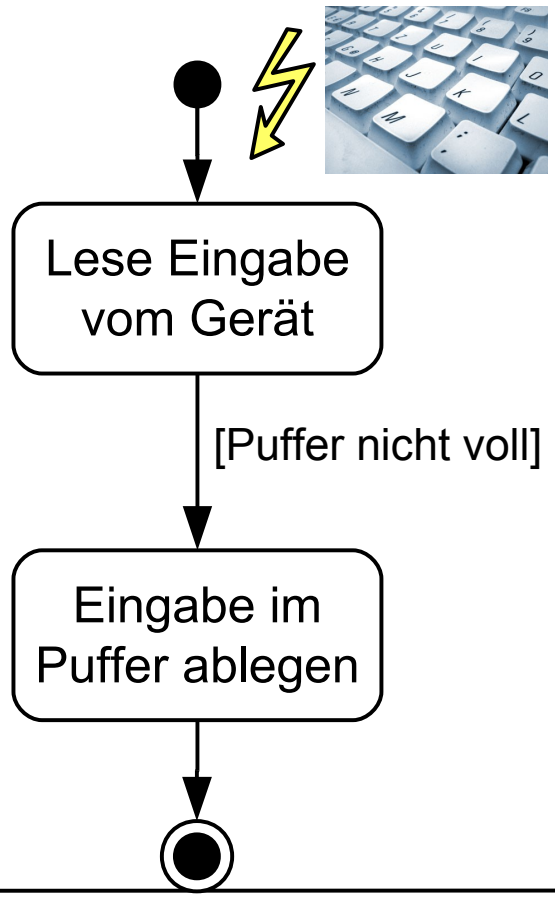
- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16 Bit CPU



Beispiel 2: Ringpuffer

- Unterbrechungen wurden eingeführt, damit das System **nicht aktiv** auf Eingaben warten muss
  - während gerechnet wird, kann die Unterbrechungsbehandlung Eingaben in einem Puffer ablegen



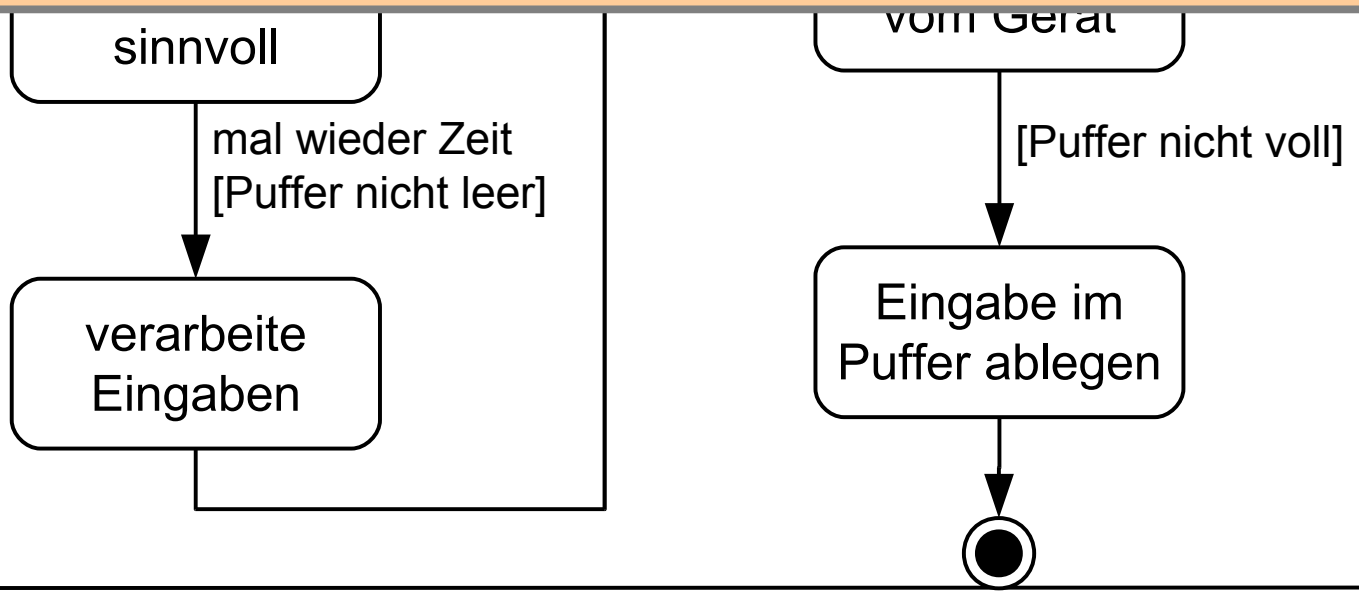


Beispiel 2: Ringpuffer

- Unterbrechungen wurden eingeführt, damit das System **nicht aktiv** auf Eingaben warten muss
  - wäh… …dlung Ein…

Problem:

Wenn die Eingabe nicht schnell genug verarbeitet werden, kann der Puffer voll werden. Die Behandlungsroutine kann die Eingabe dann nicht im Puffer ablegen. In diesem Fall geht die Eingabe verloren.



Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {          // Unterbrechungsbehandlung:
    int elements = occupied;         // Elementzähler merken
    if (elements == SIZE) return;    // Element verloren
    buf[nextin] = data;              // Element schreiben
    nextin++; nextin %= SIZE;        // Zeiger weitersetzen
    occupied = elements + 1;         // Zähler erhöhen
  }
  char consume() {                   // normaler Kontrollfluss:
    int elements = occupied;         // Elementzähler merken
    if (elements == 0) return 0;     // Puffer leer, kein Ergebnis
    char result = buf[nextout];      // Element lesen
    nextout++; nextout %= SIZE;      // Lesezeiger weitersetzen
    occupied = elements - 1;         // Zähler erniedrigen
    return result;                   // Ergebnis zurückliefern
} };
```

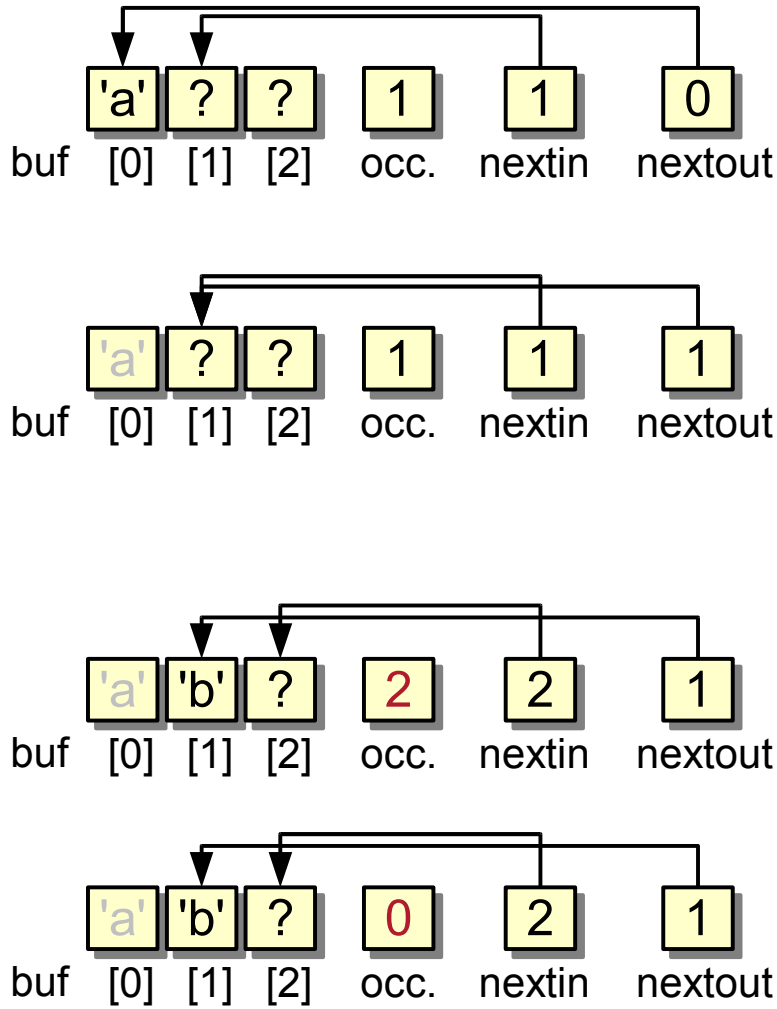
Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

Ausführung

Zustand

```
char consume() {
  int elements = occupied; // 1
  if (elements == 0) return 0;
  char result = buf[nextout]; //'a'
  nextout++; nextout %= SIZE;

void produce(char data) { // 'b'
  int elements = occupied; // 1!
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1; // 2
}

  occupied = elements - 1; // 0
  return result; // 'a'
}
```



Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

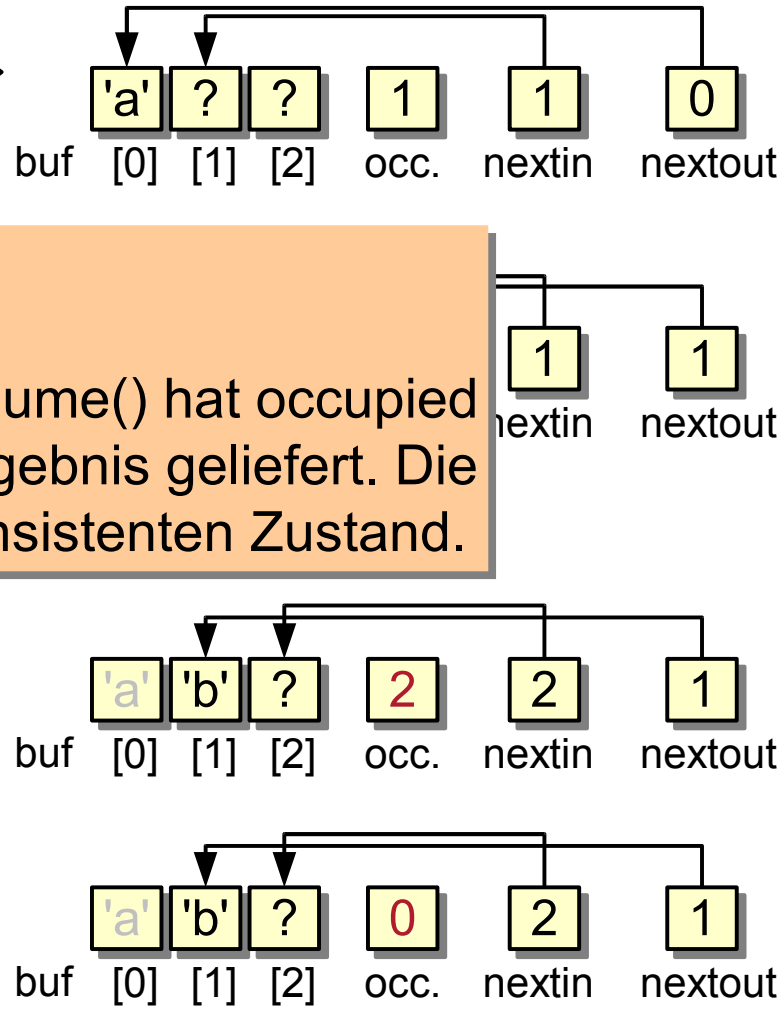
Ausführung

Zustand

```
char consume() {
  int e
  if (e
  char
  nexto

void pr
  int e
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1; // 2
}

  occupied = elements - 1; // 0
  return result; // 'a'
}
```

Problem:

Beim nächsten Aufruf von consume() hat occupied den Wert 0. Damit wird kein Ergebnis geliefert. Die Datenstruktur ist in einem inkonsistenten Zustand.



Zustandsänderung: Analyse

- selbst einzelne Zuweisungen müssen nicht atomar sein
  - Abhängigkeit vom CPU-Typ, Übersetzer und Codeoptimierung
- Pufferspeicher ist endlich
  - Behandlungsroutine kann nicht warten
  - Daten können verloren gehen
- Pufferdatenstruktur kann kaputt gehen aufgrund von ...
  - inkonsistenten Zwischenzuständen bei Änderungen durch den normalen Kontrollfluss
  - Zustandsänderungen während des Lesens (inkonsistente Kopie!)
  - Änderungen mit Hilfe einer Kopie, die nicht mehr dem Original entspricht
- das Problem ist nicht symmetrisch
  - der normale Kontrollfluss „unterbricht“ nicht die Unterbrechungsbehandlung
  - kann ausgenutzt werden!

„Harte“ Synchronisation

- Durch Unterdrückung von Unterbrechungen können *race conditions* vermieden werden:

```
char consume() {                        // normaler Kontrollfluss:
  disable_interrupts();                 // Unterbrechungen verbieten
  int elements = occupied;              // Elementzähler merken
  if (elements == 0) {                  // Puffer leer, kein Ergebnis
    enable_interrupts();                // Unterbrechungen zulassen
    return 0;
  }
  char result = buf[nextout];           // Element lesen
  nextout++; nextout %= SIZE;           // Lesezeiger weitersetzen
  occupied = elements - 1;              // Zähler erniedrigen
  enable_interrupts();                  // Unterbrechungen zulassen
  return result;                        // Ergebnis zurückliefern
}
```

- Probleme:
  - Gefahr des Verlusts von Unterbrechungsanforderungen
  - hohe und schwer vorherzusagende **„Unterbrechungslatenzen“**

Weitere Techniken in der nächsten VL

- „schlaue“ (optimistische) Verfahren
  - Datenstruktur geschickt wählen
    - möglichst wenige geteilte Elemente
    - mit weichen Konsistenzbedingungen arbeiten
  - optimistisch herangehen
    - i.d.R. tritt keine Unterbrechung im kritischen Abschnitt auf
    - falls doch, wird der Schaden festgestellt und repariert
    - ggf. wird die Operation auch wiederholt
- Pro-/Epilogmodell
  - Aufteilung der Unterbrechungsbehandlung in zwei Phasen
    - der kritische Teil wird durch einen Softwaremechanismus verzögert
    - schnelle Reaktion weiterhin möglich

Agenda

Einordnung
Begriffe und Grundannahmen
Zustandssicherung
Zustandsänderungen
Zusammenfassung

Zusammenfassung

- die korrekte Behandlung von Unterbrechungen gehört zu den schwierigsten Aufgaben im Betriebssystembau
  - Quelle der Asynchronität
    - gleichzeitig Segen und Fluch
  - Zustandssicherung auf Registerebene
    - Assemblerprogrammierung!
    - Abhängigkeit vom Übersetzer (z.B. flüchtige/nicht-flüchtige Register)
  - unterschiedliche Modelle (Prioritäten, u.s.w.)
- Zustandsänderungen in der Unterbrechungsbehandlung müssen wohl überlegt sein
  - kritische Abschnitte schützen
  - Fehler schwer zu finden (nicht verlässlich reproduzierbar!)

Betriebssysteme (BS)

VL 5 – Unterbrechungen, Software

Volkmar Sieh / **Daniel Lohmann**

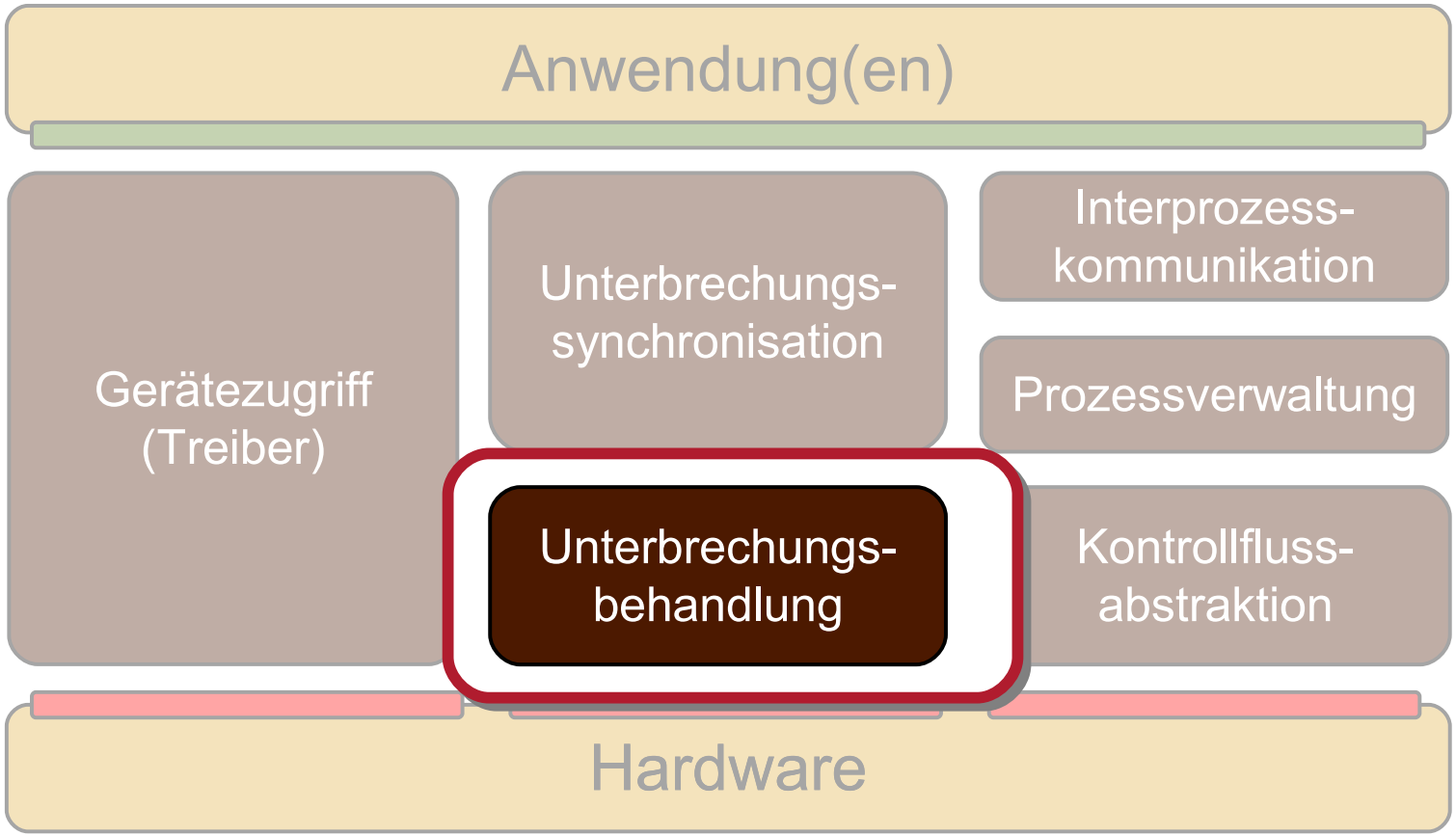
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 22. November 2023

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

Beispiel Console

Was passiert beim Tippen eines Zeichens in der Console?

1. Benutzer tippt Taste.
2. Keyboard-Controller signalisiert Interrupt.
3. CPU liest Key-Code aus.
4. CPU übersetzt Key-Code in ASCII-Zeichen.
5. ASCII-Zeichen wird im Eingabe-Puffer abgelegt.
6. Wenn „ECHO”-Modus eingeschaltet ist, wird Zeichen auf Bildschirm dargestellt (evntl. Scrollen, Cursor verschieben, Pixel gemäß Zeichensatz setzen/löschen).
7. Die CPU weckt ggf. einen wartenden Prozess auf.

Beispiel Netzwerk

Was passiert beim Empfang einer Ethernet-TCP-Nachricht?

1. Eine Netzwerk-Karte empfängt das Ethernet-Paket.
2. Die Netzwerk-Karte signalisiert einen Interrupt.
3. Die CPU leitet das Paket z.B. an die IP-Schicht weiter.
4. CPU überprüft die IP-Check-Summe.
5. In der IP-Schicht werden ggf. mehrere IP-Pakete zu einem Gesamt-Paket zusammengesetzt.
6. Die CPU leitet das Paket z.B. an die TCP-Schicht weiter.
7. In der TCP-Schicht werden z.B. ACK-Nachrichten generiert.
8. Die Daten werden in die Warteschlange des entsprechenden Ports eingehängt.
9. Die CPU weckt ggf. einen wartenden Prozess auf.

Beispiel NFS-Server

Was passiert, wenn NFS-Client beim Server Daten anfordert?

1. IP-Nachrichtenempfang s.o.
2. IP-Nachricht wird in die UDP-Schicht weitergeleitet.
3. Gemäß dem Auftrag im Paket werden z.B. von Platte Daten gelesen.
4. Daten und Ziel-/Empfänger-Adresse werden in ein neues UDP-Paket kopiert.
5. UDP-Paket wird in IP-Schicht weitergeleitet.
6. IP-Paket wird ggf. in mehrere kleinere IP-Paket zerteilt.
7. Für jedes Paket wird eine Check-Summe erstellt.
8. Alle IP-Pakete werden in die Ethernet-Schicht weitergeleitet.
9. Netzwerk-Karte verschickt die Ethernet-Pakete.

Hardware-Unterbrechungen

Probleme:

- Während der ganzen Unterbrechungslaufzeit sind alle weiteren bzw. alle niederprioren Unterbrechungen gesperrt.
  - Gefahr von großer Interrupt-Verzögerung
  - Gefahr von Datenverlusten
- Unterbrechungsbehandlung kann nicht passiv warten.
  - Interrupts sind gesperrt

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

Beispiel Console

Was passiert beim Tippen eines Zeichens in der Console?

1. Benutzer tippt Taste.
2. Keyboard-Controller signalisiert Interrupt.
3. CPU liest Key-Code aus.

   <= Hardware-IRQ abgearbeitet!
4. CPU übersetzt Key-Code in ASCII-Zeichen.
5. ...

   Nach Punkt 3 könnten IRQs wieder erlaubt werden!

Aufteilung Unterbrechungsbehandlung

Unterbrechungsbehandlung zweigeteilt:

1. Teil, der Zeichen/Pakete/... bei der Hardware ausliest und in einen Puffer kopiert.
   - Interagiert nur minimal mit dem Rest des Systems.
   - Kann (fast) immer ablaufen.
2. Teil, der Zeichen/Pakete/... aus Puffer ausliest und weiterverarbeitet.
   - Ist weitgehend Hardware-unabhängig.
   - Kann (fast) immer unterbrochen werden.

Aufteilung Unterbrechungsbehandlung

Unterbrechungsbehandlung zweigeteilt:

1. Teil ggf. leer

Beispiel: Timer/Uhr-Interrupt

(Typisch: alle „Edge-Triggered”-Interrupts)

Aufteilung Unterbrechungsbehandlung

Multiprozessoren:

- 1. und 2. Teil können auf verschiedenen Kernen ausgeführt werden (Beispiel: E/A-Interrupts; Last-Verteilung).

  oder

- 1. und 2. Teil müssen auf gleichem Kern ausgeführt werden (Beispiel: Schedule-Interrupt).

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

Beispiel / Idee

```
void keyboard_hard_irq() {
        uint8_t code = read_code_from_keyboard_controller();
        if (count < sizeof(buffer)) {
                buffer[head] = code;
                head = (head + 1) % sizeof(buffer);
                count++;
        }
}
void keyboard_soft_irq() {
        asm volatile ("cli\n");
        while (1 <= count) {
                uint8_t code = buffer[tail];
                tail = (tail + 1) % sizeof(buffer);
                count--;
                asm volatile ("sti\n");
                process_code(code);
                asm volatile ("cli\n");
        }
        asm volatile ("sti\n");
}
void keyboard_irq() {
        keyboard_hard_irq();
        asm volatile ("sti\n");
        keyboard_soft_irq();
}
```

Beispiel / Idee

Funktioniert so eventuell nicht!

```
void keybo
        keyboard_hard_irq();
        asm volatile ("sti\n");
        keyboard_soft_irq();
}
```

- Problem: nach `sti` kann Stack überlaufen!
- Daher (ähnlich Linux):

```
void keyboard_irq() {
        keyboard_hard_irq();
        apic_disable_keyboard_irq();
        asm volatile ("sti\n");
        keyboard_soft_irq();
        asm volatile ("cli\n");
        apic_enable_keyboard_irq();
}
```

SoftIRQs

```
apic_enable_keyboard_irq();
```

```
apic_disable_keyboard_irq();
```

können ggf. alle Interrupts nierigerer Priorität ein- bzw. ausschalten.

=> selbstdefinierte Interrupt-Prioritäten

Aufteilung Unterbrechungsbehandlung

Nachteile:

- Schreiben in / Lesen aus Puffer kostet Zeit.
- Disable/Enable von Interrupts kostet Zeit.

Vorteil:

- Andere(!), ggf. wichtige(!), Interrupts werden früher bearbeitet.

Shared Interrupts

Im Falle von „shared interrupts” müssen

- erst von allen beteiligten Geräten ggf. Daten ausgelesen und zwischengespeichert,
- dann Interrupts wieder zugelassen,
- dann von allen beteiligten Geräten die zwischengespeicherten Daten weiterverarbeitet werden

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

SoftIRQs

Hinweis: `keyboard_soft_irq` muss ggf. gar nicht aufgerufen werden:

- Eine Taste wurde losgelassen.
- Eine Shift/Alt/Ctrl/...-Taste wurde gedrückt.

Daher könnte `keyboard_hard_irq`-Methode Bit setzen, wenn `keyboard_soft_irq`-Methode ausgeführt werden soll.

=> Entspricht einem Interrupt-Mechanismus
(diesmal in Software)
sogenannte „**SoftIRQs**”

SoftIRQ-Interface

Typisches SoftIRQ-Interface (pro SoftIRQ):

`configure:` Registriere Behandlungsfunktion, die beim Auftreten des SoftIRQs aufgerufen werden soll.

`trigger:` Wenn SoftIRQ disabled: setze Pending-Bit, sonst: führe Behandlungsfunktion aus.

`disable:` Disable SoftIRQ.

`enable:` Enable SoftIRQ. Wenn Pending-Bit gesetzt, führe Behandlungsfunktion aus.

SoftIRQ-Interface

Vorsicht! Race-Condition:

```
void disable() {
        enabled = false;
}

void enable() {
        while (pending) {
                pending = false;
                handler();
        }

        enabled = true;
}

void trigger() {
        if (enabled) {
                asm volatile ("sti\n");
                handler();
                asm volatile ("cli\n");
        } else {
                pending = true;
        }
}
```

<= trigger hat keinen Effekt!

SoftIRQ-Interface

`trigger` wird nur im Interrupt-Handler aufgerufen.
Kann daher mit `cli`/`sti` hart synchronisiert werden:

```
void enable() {
        asm volatile("cli\n");
        while (pending) {
                pending = false;
                asm volatile("sti\n");
                handler();
                asm volatile("cli\n");
        }
        enabled = true;
        asm volatile("sti\n");
}
```

SoftIRQ-Interface

Idee: *BSD: `trigger` wird nur im Interrupt-Handler aufgerufen und kennt damit gesicherten Instruktion-Pointer (IP).

```
void enable() {
        asm volatile("enable_begin:\n");
        while (pending) {
                pending = false;
                handler();
        }
        enabled = true;
        asm volatile("enable_end:\n");
}

void trigger() {
        if (enabled) {
                asm volatile("sti\n");
                handler();
                asm volatile("cli\n");
        } else {
                pending = 1;
                if (enable_begin <= IP && IP <= enable_end) {
                        IP = enable_begin;
                }
        }
}
```

SoftIRQ-Interface

Hardware-IRQs werden beim Signalisieren eines Interrupts von der Hardware aus nur dann gestartet, wenn sie nicht schon laufen.

Kann durch weiteres Flag (`running`) nachgebildet werden.

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

SoftIRQ-Beispiel

```
void keyboard_hard_irq() {
        uint8_t code = read_code_from_keyboard_controller();
        if (count < sizeof(buffer)) {
                buffer[head] = code;
                head = (head + 1) % sizeof(buffer);
                count++;
        }
        apic_disable_keyboard_irq();
        trigger();
        apic_enable_keyboard_irq();
}

void keyboard_soft_irq() {
        asm volatile ("cli\n");
        while (0 < count) {
                uint8_t code = buffer[tail];
                tail = (tail + 1) % sizeof(buffer);
                count--;
                asm volatile ("sti\n");
                print_to_screen(code);
                asm volatile ("cli\n");
        }
        asm volatile ("sti\n");
}
```

SoftIRQ-Beispiel

```
void keyboard_soft_irq() {
        asm volatile ("cli\n");
        while (0 < count) {
                uint8_t code = buffer[tail];
                tail = (tail + 1) % sizeof(buffer);
                count--;
                asm volatile ("sti\n");
                print_to_screen(code);
                asm volatile ("cli\n");
        }
        asm volatile ("sti\n");
}

void console_write(char code) {
        disable();
        print_to_screen(code);
        enable();
}
```

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

Verwandte Konzepte – Top/Bottom Halves

Idee:

- Interrupt-Handler liest Daten aus Gerät in Puffer und setzt Bit
- vor Rückkehr in den User-Mode werden Puffer ausgelesen und die Daten verarbeitet, wenn Bit gesetzt

(Linux)

Verwandte Konzepte – ksoftirqd

Idee:

- `while (pending) { pending = false; handler(); }`-Schleife wird nur begrenzte Anzahl von Malen durchlaufen.
- Danach wird „ksoftirqd” (Kernel-Thread) aufgeweckt, der SoftIRQ-Abarbeitung übernimmt.

Vorteil: Interrupts können CPUs nicht monopolisieren. (Linux)

Verwandte Konzepte – tasklets

Idee:
Hard-Interrupt-Handler triggert nicht (per Nummer) die Ausführung einer bestimmten Methode,
sondern fügt eine beliebige Methode in eine Liste ein.
Liste wird dann vor Rückkehr in User-Mode abgearbeitet.
(Linux)

Verwandte Konzepte – Interrupt Threads

Idee:

- Interrupt-Handler liest Daten aus Gerät in Puffer
- weckt Thread auf
- Thread liest Puffer aus und ...

Threads können Prioritäten besitzen.
Threads können warten.

Verwandte Konzepte – Interrupt Workqueues

Idee:

- Interrupt-Handler fügen Aufträge in Work-Queue ein.
- Work-Queue wird von Worker-Thread-Pool abgearbeitet.

Agenda

Einordnung
Beispiele
Aufteilung Interrupt-Bearbeitung
Implementierung
SoftIRQs
SoftIRQ-Beispiel
Verwandte Konzepte
Zusammenfassung

Zusammenfassung

- Viele verschiedene Möglichkeiten, Interrupts abzuarbeiten.
- Immer Ziel: schnelle Reaktion auf (wichtige) Interrupts.
- Fast immer: „Hard”- und „Soft”-Interrupt-Teil.
- SoftIRQs: „Interrupts” in Software.
- In aktuellen BS existieren weitere, von SoftIRQs abgeleitete Konzepte.

Betriebssysteme (BS)

VL 6 – Unterbrechungen, Synchronisation

Volkmar Sieh / **Daniel Lohmann**

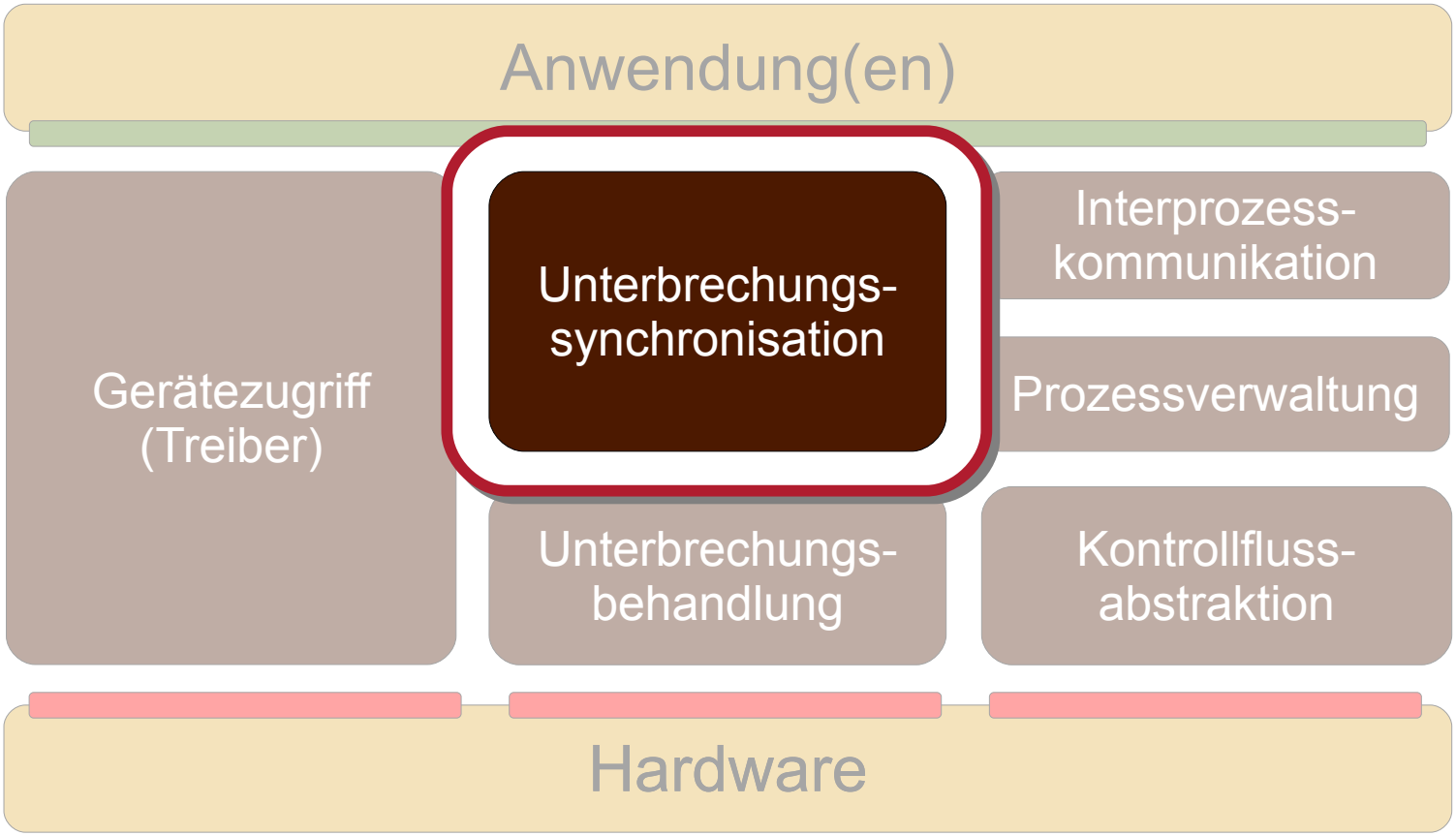
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 29. November 2023

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

Einleitung
Prioritätsebenenmodell
Harte Synchronisation
Weiche Synchronisation
Prolog/Epilog-Modell
Zusammenfassung
Referenzen

Agenda

Einleitung
- Motivation
- Erstes Fazit

Prioritätsebenenmodell

Harte Synchronisation

Weiche Synchronisation

Prolog/Epilog-Modell

Zusammenfassung

Referenzen

Motivation: Konsistenzprobleme

Zustandsänderungen ...

- sind Sinn und Zweck der Unterbrechungsbehandlung
  - Gerätetreiber müssen über den Abschluss einer E/A Operation informiert werden
  - der Scheduler muss erfahren, dass eine Zeitscheibe abgelaufen ist
- müssen mit Vorsicht durchgeführt werden
  - Unterbrechungen können zu jeder Zeit auftreten
  - kritisch sind Daten/Datenstrukturen, die der normale Kontrollfluss und die Unterbrechungsbehandlung sich teilen

Beispiele aus der letzten Vorlesung

Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

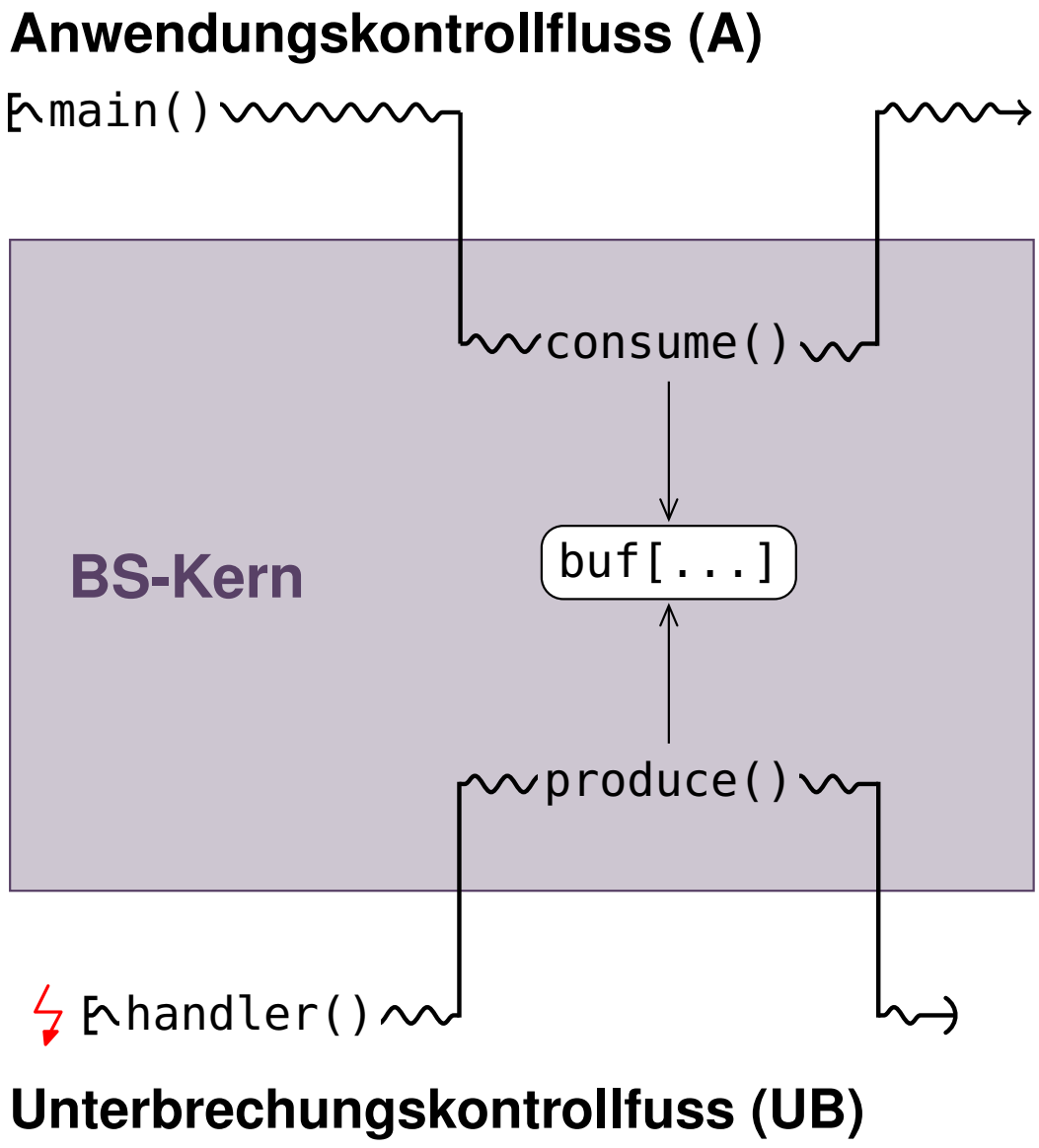
```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {        // Unterbrechungsbehandlung:
    int elements = occupied;       // Elementzähler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zähler erhöhen
  }
  char consume() {                 // normaler Kontrollfluss:
    int elements = occupied;       // Elementzähler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zähler erniedrigen
    return result;                 // Ergebnis zurückliefern
} };
```

Motivation: Ursache

Kontrollflüsse
“von oben”

“begegnen”
sich im Kern

und “von unten”

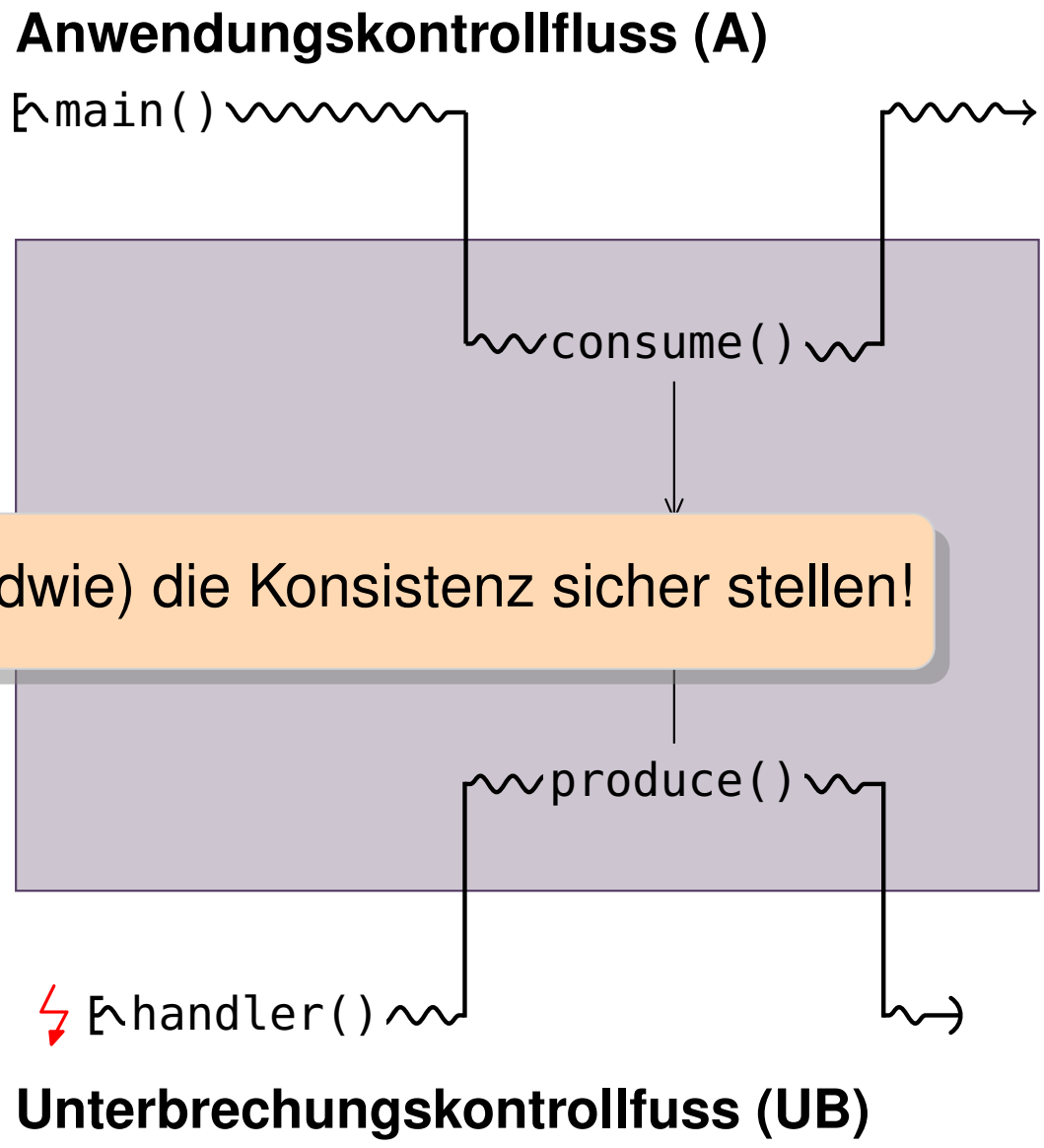


Motivation: Ursache

Kontrollflüsse
“von oben”

“b
si

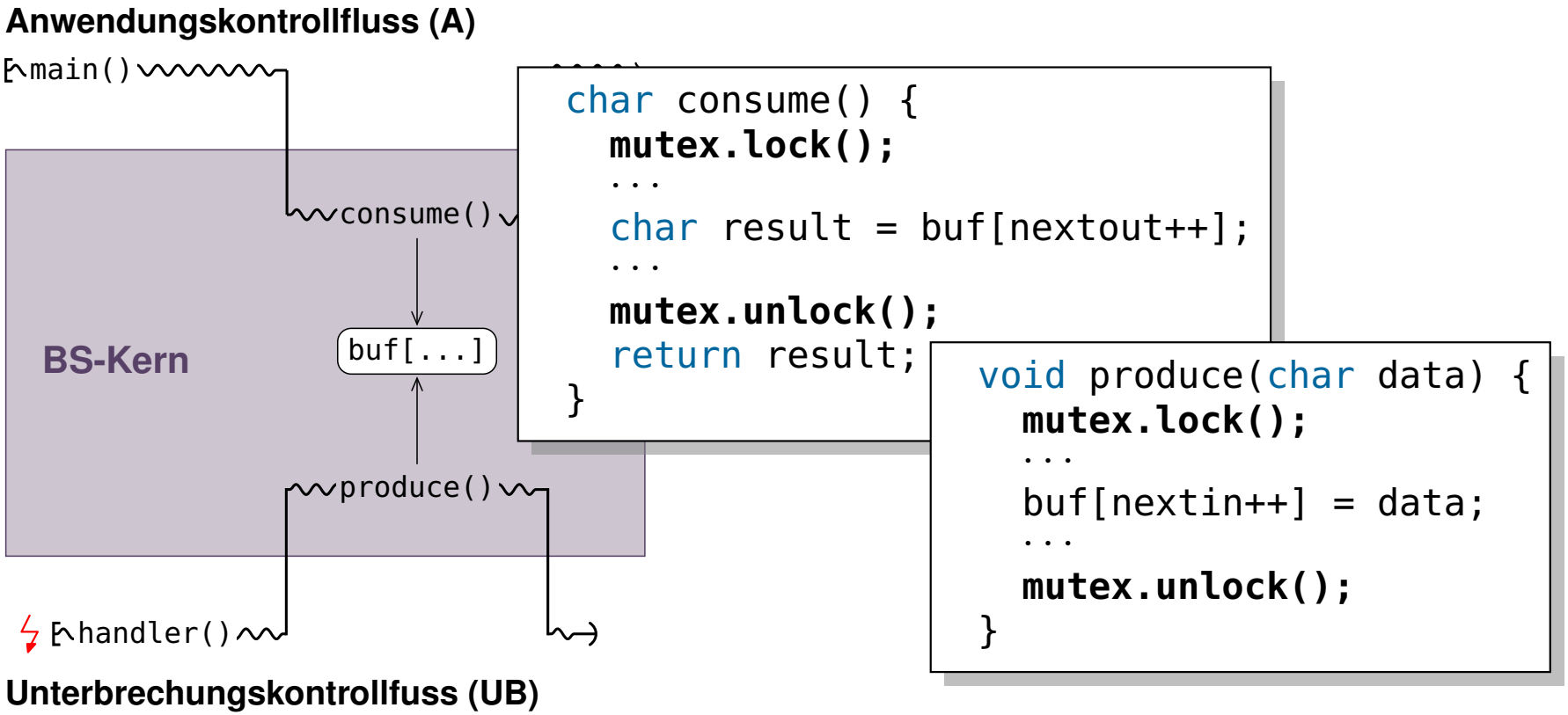
und “von unten”



Wir müssen (irgendwie) die Konsistenz sicher stellen!

Naiver Lösungsansatz

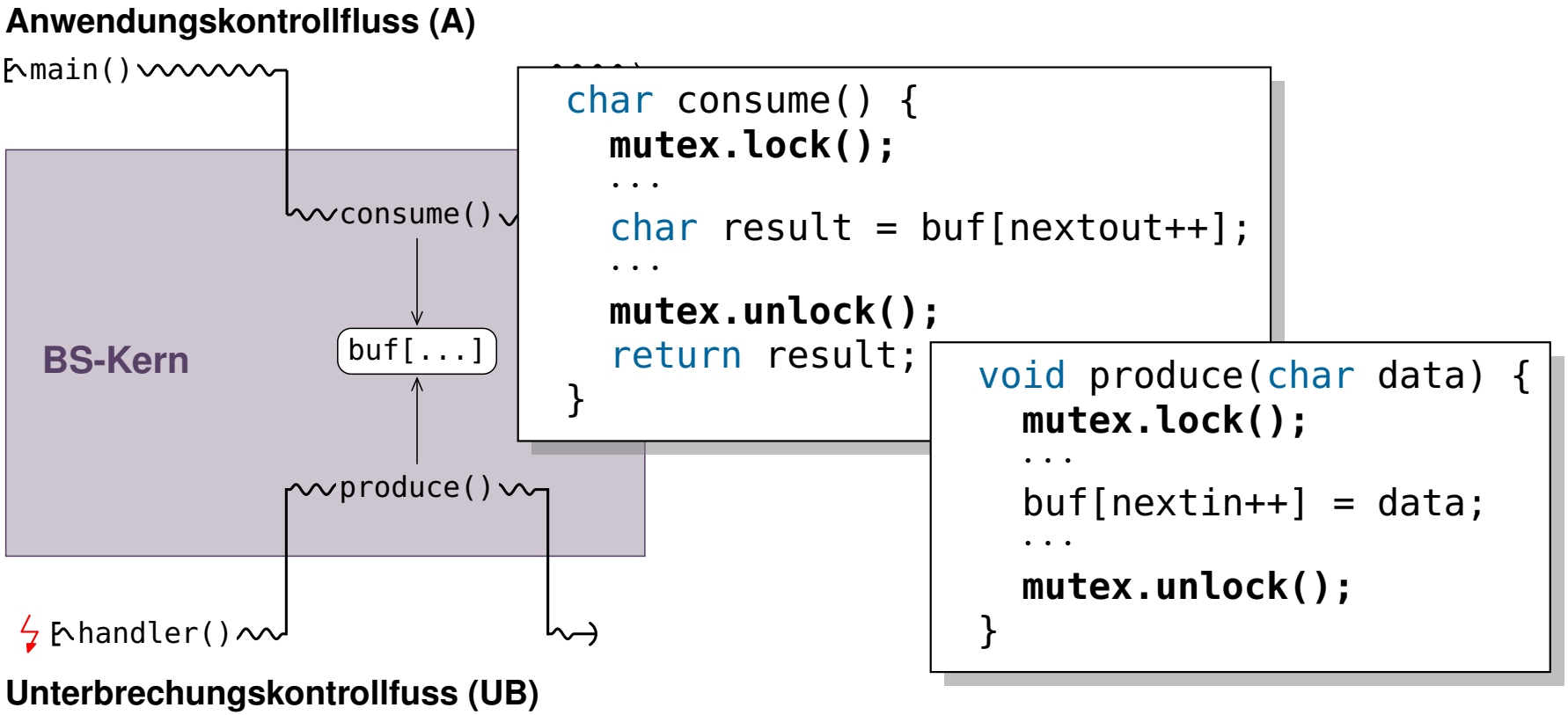
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch Mutex, *Spin-Lock*, . . . (vgl. [SP])
  - wie zwischen zwei Prozessen



Naiver Lösungsansatz

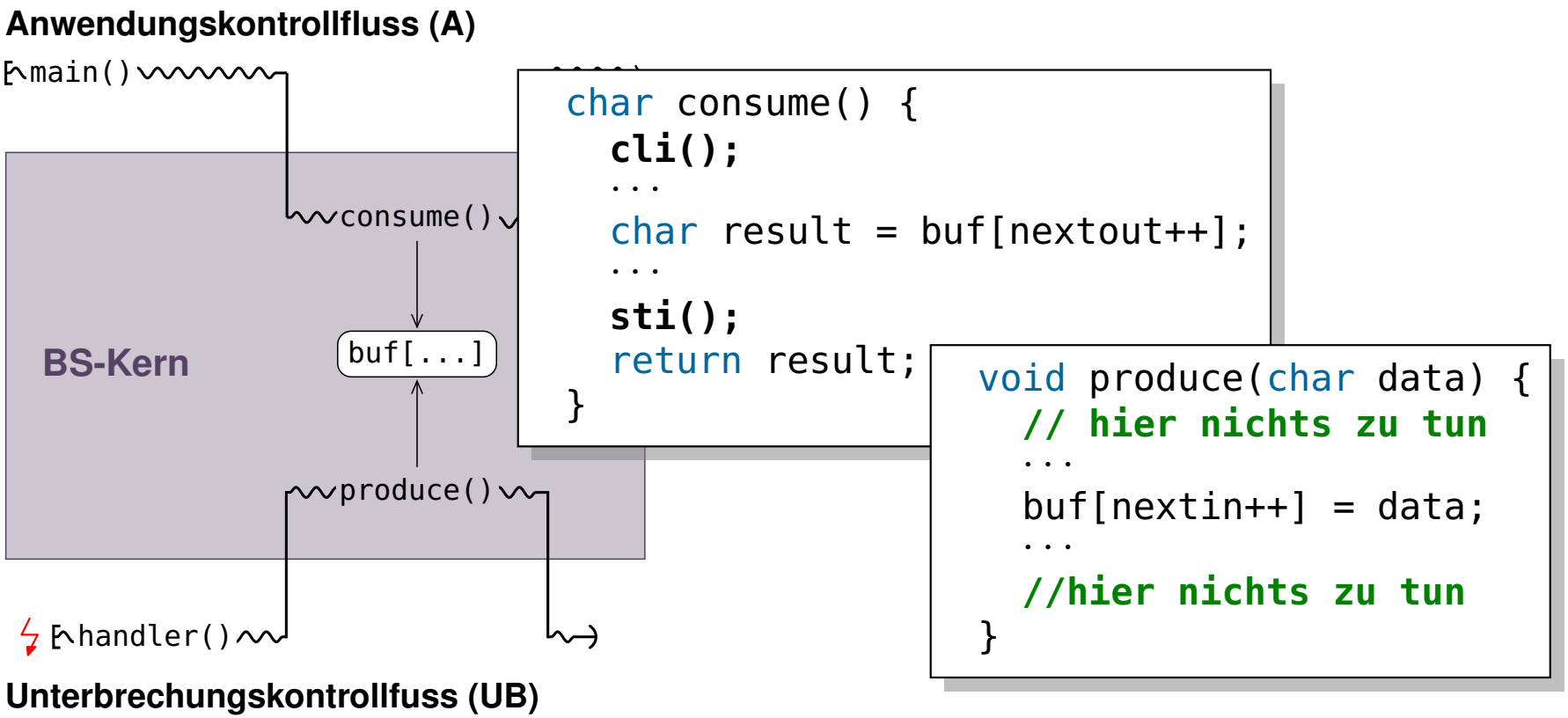
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch M
  - wie zwischen zwei Prozessen

Zweiseitige Synchronisation funktioniert **natürlich nicht!**



Besserer Lösungsansatz

- Einseitige Synchronisation
  - Unterdrückung der Unterbrechungsbehandlung im Verbraucher
  - Operationen `disable_interrupts()` `enable_interrupts()`
    (im Folgenden o. B. d. A. in „Intel“-Schreibweise: `cli()` / `sti()`)

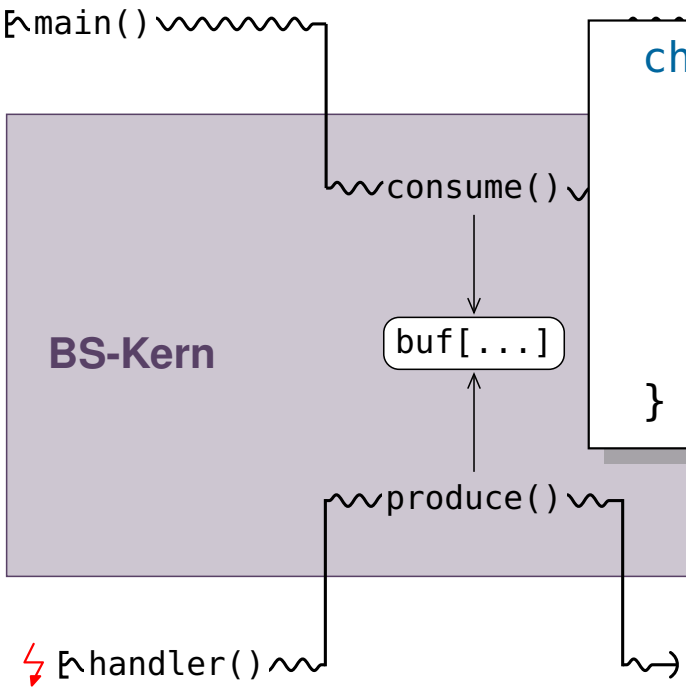


Besserer Lösungsansatz

- Einseitige Synchronisation
  - Unterdrückung der Unterbrechungs
  - Operationen `disable_interrupts(`
    (im Folgenden o. B. d. A. in „Intel“-Schreibw

Einseitige Synchronisation funktioniert. [Warum?]

Anwendungskontrollfluss (A)




```
char consume() {
  cli();
  ...
  char result = buf[nextout++];
  ...
  sti();
  return result;
}
```

```
void produce(char data) {
  // hier nichts zu tun
  ...
  buf[nextin++] = data;
  ...
  //hier nichts zu tun
}
```

Unterbrechungskontrollfuss (UB)

Erstes Fazit

- Konsistenzsicherung zwischen
  - Anwendungskontrollfluss (A) und
  - Unterbrechungsbehandlung (UB)

  muss anders erfolgen als zwischen Prozessen
- Die Beziehung zwischen A und UB ist **asymmetrisch**
  - Es handelt sich um „verschiedene Arten“ von Kontrollflüssen
  - UB *unterbricht* Anwendungskontrollfluss
    - implizit, an beliebiger Stelle
    - hat immer Priorität, läuft durch (*run-to-completion*)
  - A kann UB *unterdrücken* (besser: *verzögern*)
    - explizit, mit `cli`/`sti` (Grundannahme 5 aus VL 4)
- Synchronisation / Konsistenzsicherung erfolgt **einseitig**

Diese Tatsachen müssen wir **beachten**!

(Das heißt aber auch: Wir können sie **ausnutzen**)

Agenda

Einleitung

Prioritätsebenenmodell

- Grundbegriffe
- Verallgemeinerung
- Konsistenzsicherung

Harte Synchronisation

Weiche Synchronisation

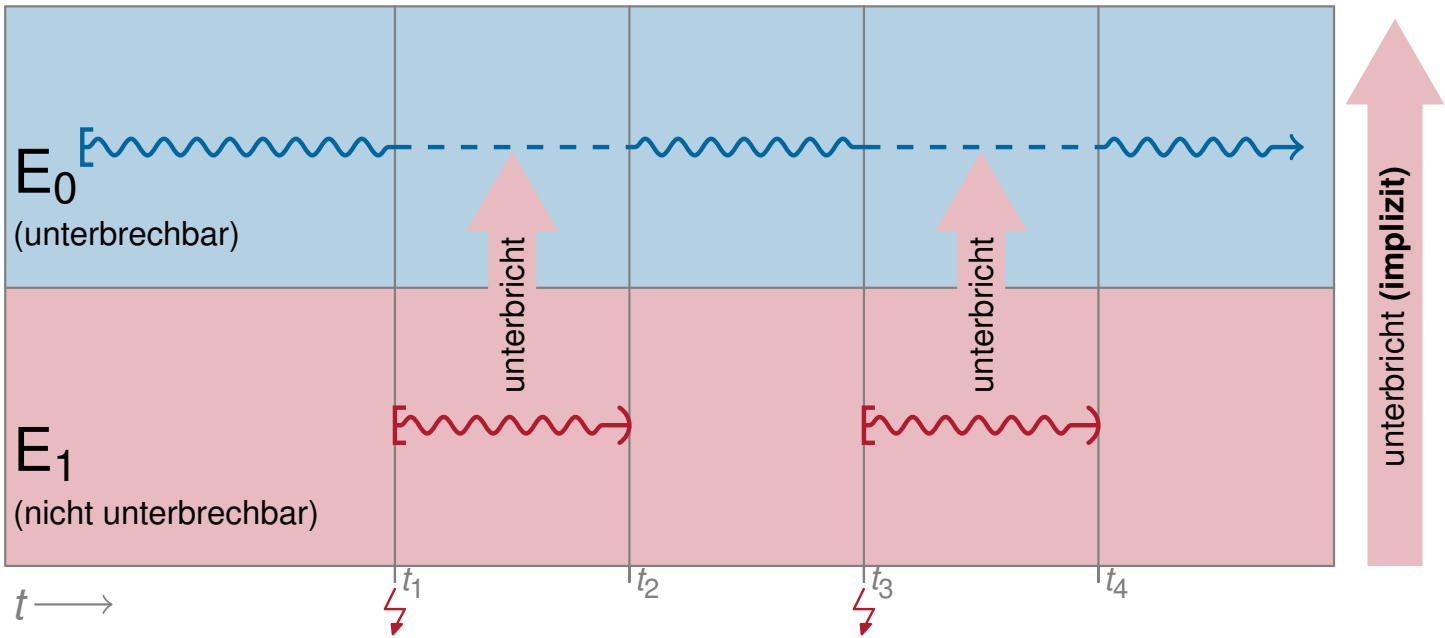
Prolog/Epilog-Modell

Zusammenfassung

Referenzen

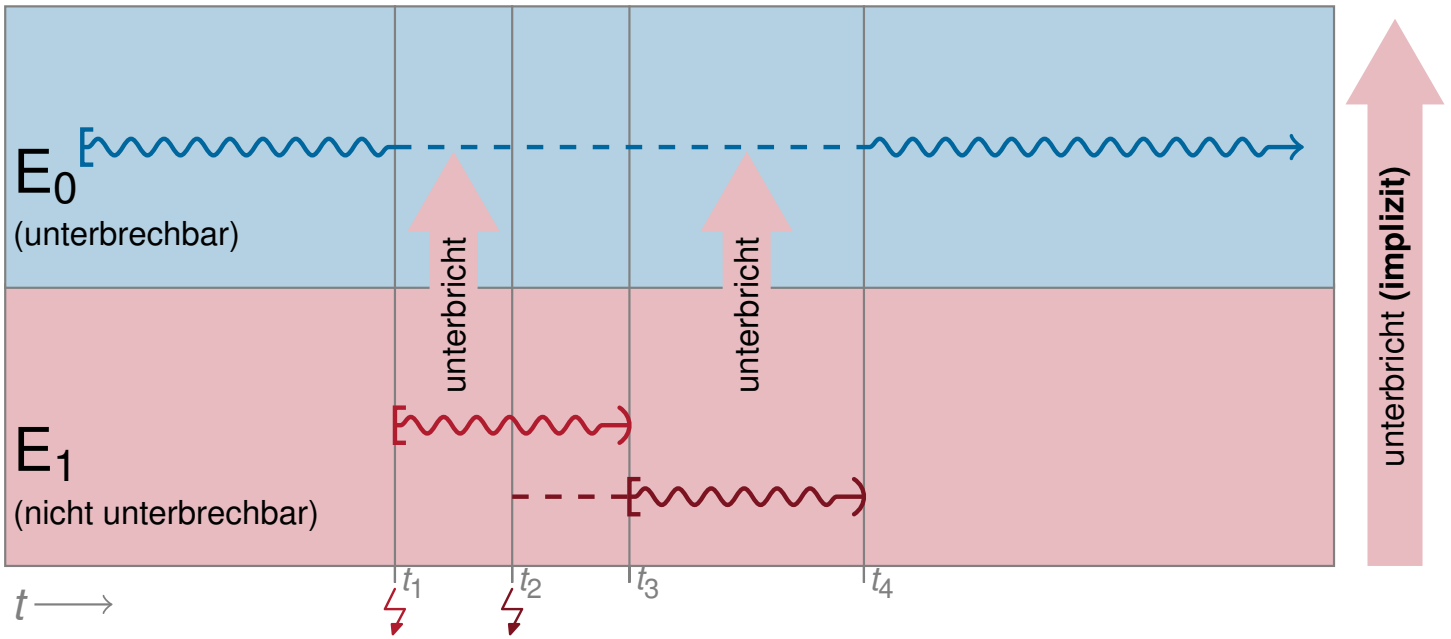
Prioritätsebenenmodell

- E_0 sei die Anwendungskontrollfluss-Ebene (A)
  - Kontrollflüsse dieser Ebene sind jederzeit unterbrechbar (durch E_1-Kontrollflüsse, implizit)
- E_1 sei die Unterbrechungsbehandlungs-Ebene (UB)
  - Kontrollflüsse dieser Ebene sind nicht unterbrechbar (durch $E_{0/1}$-Kontrollflüsse, implizit)



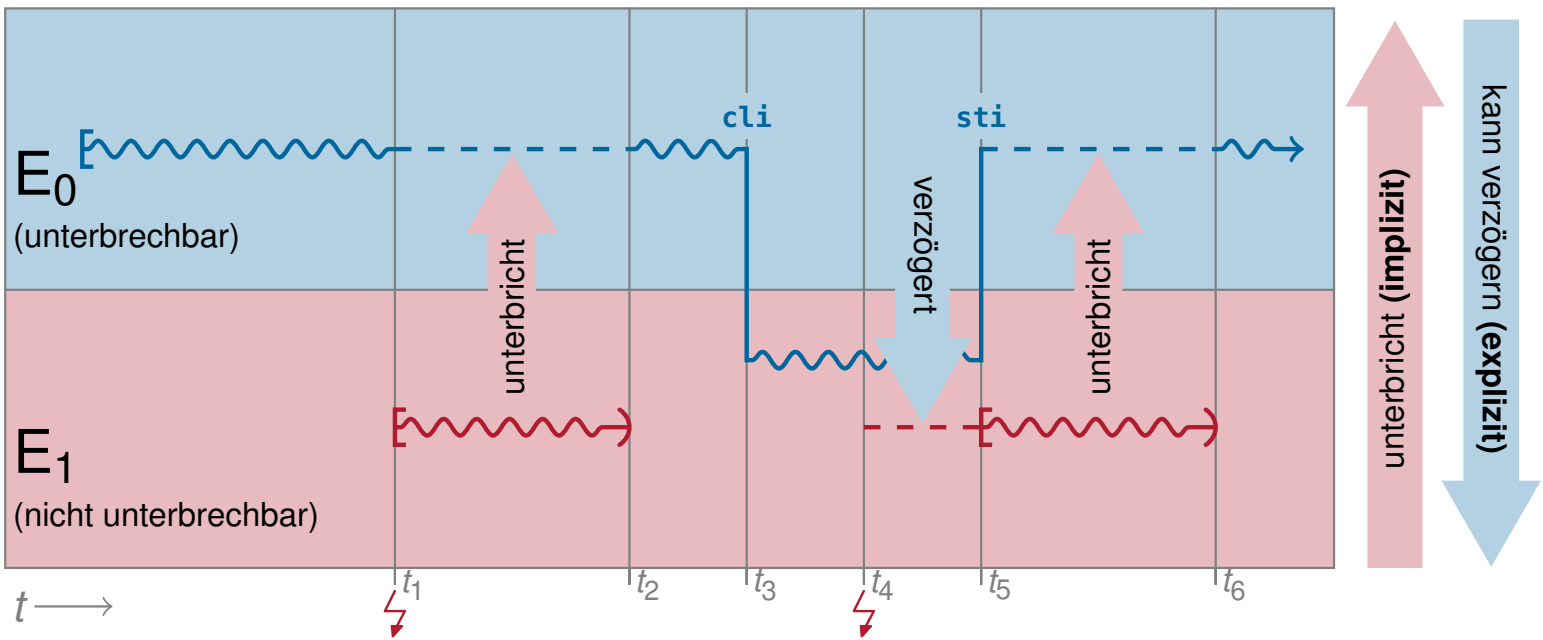
Prioritätsebenenmodell

- Kontrollflüsse derselben Ebene werden sequentialisiert
  - Sind mehrere Kontrollflüsse in einer Ebene anhängig, so werden diese nacheinander abgearbeitet (*run-to-completion*)
    - damit ist auf jeder Ebene höchstens ein Kontrollfluss aktiv
  - Die Sequentialisierungsstrategie selber ist dabei beliebig
    - FIFO, LIFO, nach Priorität, zufällig, . . .
    - Für E_1-Kontrollflüsse auf dem PC implementiert der (A)PIC die Strategie



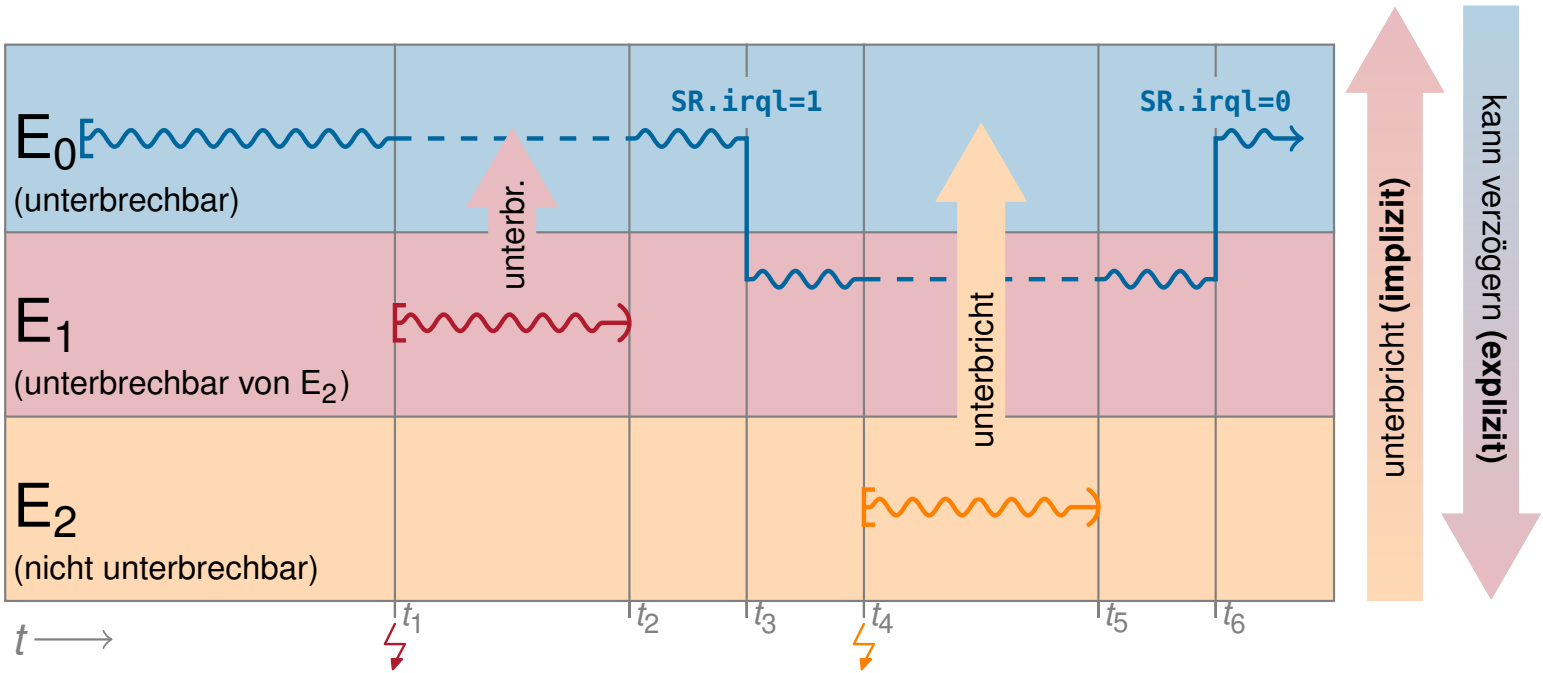
Prioritätsebenenmodell

- Kontrollflüsse können die Ebene wechseln
  - Mit `cli` wechselt ein E_0-Kontrollfluss explizit auf E_1
    - er ist ab dann nicht mehr unterbrechbar
    - andere E_1-Kontrollflüsse werden verzögert (↩ Sequentialisierung)
  - Mit `sti` wechselt ein E_1-Kontrollfluss explizit auf E_0
    - er ist ab dann (wieder) unterbrechbar
    - anhängige E_1-Kontrollflüsse „schlagen durch“ (↩ Sequentialisierung)



Prioritätsebenenmodell

- Verallgemeinerung für mehrere Unterbrechungsebenen:
  - Kontrollflüsse auf E_l werden
    1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
    2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
    3. sequentialisiert mit weiteren Kontrollflüssen von E_l
  - Kontrollflüsse können die Ebene wechseln
    - durche spezielle Operationen (hier: Modifizieren des Statusregisters)



Prioritätsebenenmodell: Konsistenzsicherung

- Jede Zustandsvariable ist (logisch) genau einer Ebene E_l zugeordnet
  - Zugriffe aus E_l sind implizit konsistent ($\hookleftarrow$ Sequentialisierung)
  - Konsistenz bei Zugriff aus höheren / tieferen Ebenen muss explizit sichergestellt werden
- Maßnahmen zur Konsistenzsicherung bei Zugriffen:
  - „von oben“ (aus E_k mit $k < l$) durch harte Synchronisation
    - **explizit die Ebene auf E_l wechseln** beim Zugriff (Verzögerung)
    - damit erfolgt der Zugriff aus derselben Ebene ($\hookrightarrow$ Sequentialisierung)
  - „von unten“ (aus E_m mit $m > l$) durch weiche Synchronisation
    - **algorithmisch sicherstellen**, dass Unterbrechungen nicht stören
    - erfordert unterbrechungstransparente Algorithmen

Agenda

Einleitung
Prioritätsebenenmodell
Harte Synchronisation
- Ansatz
- Bewertung

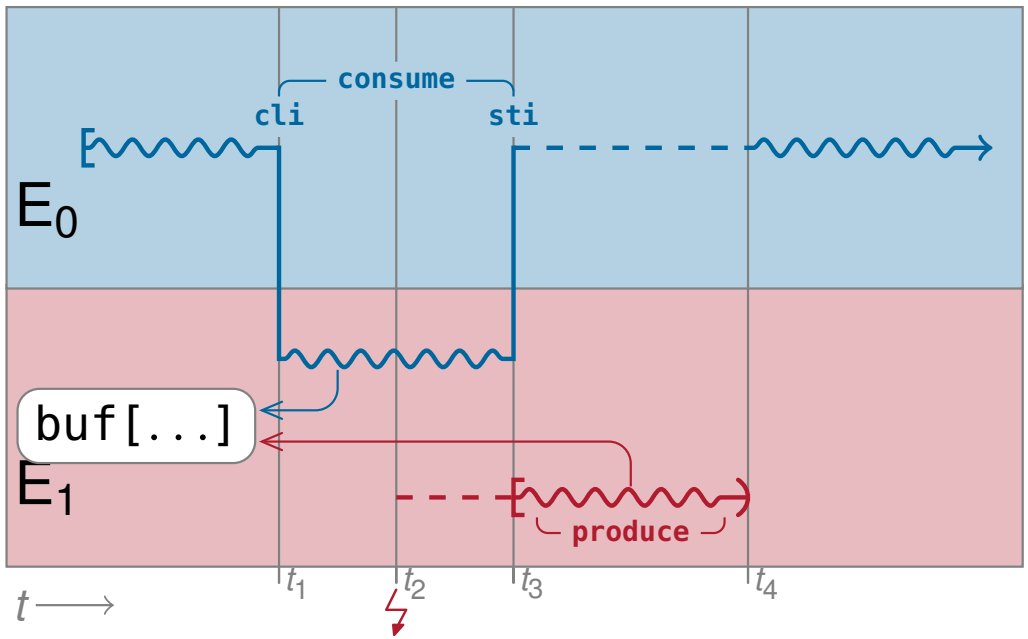
Weiche Synchronisation
Prolog/Epilog-Modell
Zusammenfassung
Referenzen

Bounded Buffer – Lösung mit harter Synchronisation

Zugriff „von oben“ wird hart synchronisiert: Für die Ausführung von `consume()` wechselt der Kontrollfluss auf E_1

```
char consume() {
  cli();
  ...
  char result = buf[nextout++];
  ...
  sti();
  return result;
}
```

```
void produce(char data) {
  // hier nichts zu tun
  ...
  buf[nextin++] = data;
  ...
  //hier nichts zu tun
}
```



Zustand liegt (logisch) auf E_1

Harte Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sicher gestellt
    - auch bei komplexen Datenstrukturen und Zugriffsmustern
    - unabhängig davon, was der Compiler macht
  - einfach anzuwenden, „funktioniert immer“
    - im Zweifelsfall legt man einfach sämtlichen Zustand auf die höchstpriore Ebene
- **Nachteile**
  - Breitbandwirkung
    - Es werden pauschal alle Unterbrechungsbehandlungen (Kontrollflüsse) auf und unterhalb der Zustandsebene verzögert
  - Prioritätsverletzung
    - Es werden Kontrollflüsse höherer Priorität verzögert
  - prophylaktisches Verfahren
    - Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine relevante Unterbrechung eintrifft, sehr klein ist.

Harte Synchronisation: Bewertung (Forts.)

- Ob die Nachteile erheblich sind, hängt ab von
  - Häufigkeit,
  - durchschnittlicher Dauer,
  - maximaler Dauer

  der Verzögerung.
- Kritisch ist vor allem die **maximale Dauer**
  - hat direkten Einfluss auf die anzunehmende Latenz
  - Wird die Latenz zu hoch, können Daten verloren gehen
    - *edge-triggered* Unterbrechungen gehen verloren
    - Daten werden zu langsam von EA-Gerät abgeholt

Fazit

Harte Synchronisation ist eher **ungeeignet** für die Konsistenzsicherung **komplexer Datenstrukturen**

Agenda

Einleitung
Prioritätsebenenmodell
Harte Synchronisation
Weiche Synchronisation
- Ansatz
- Implementierungsbeispiele
- Bewertung

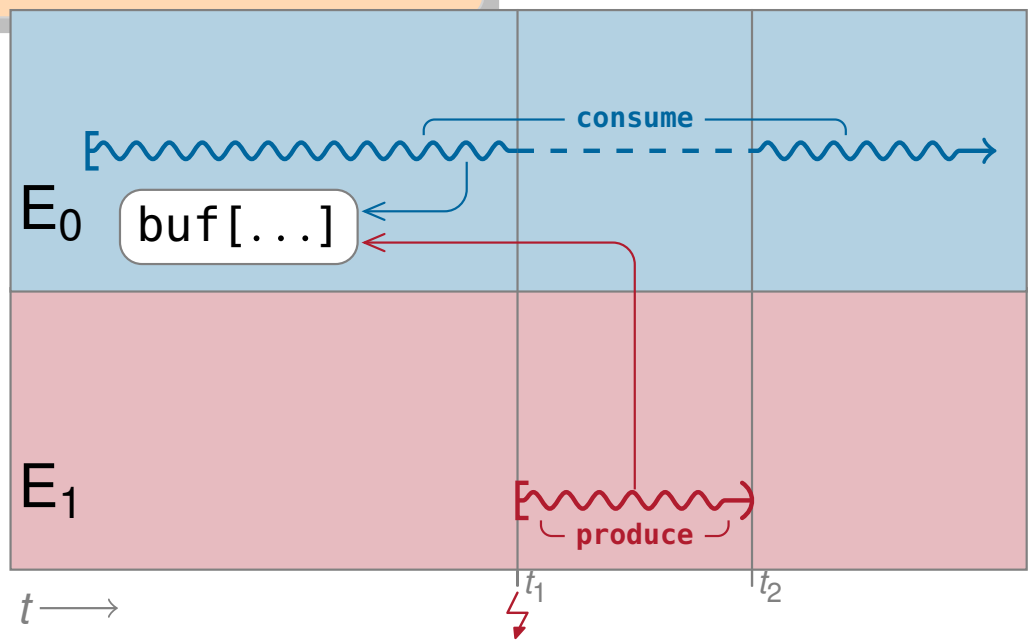
Prolog/Epilog-Modell
Zusammenfassung
Referenzen

Bounded Buffer – Ansatz mit weicher Synchronisation

Zugriff „von unten“ wird weich synchronisiert: `consume()` liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung `produce()` ausgeführt wurde.

```
char consume() {

      ?

}
```

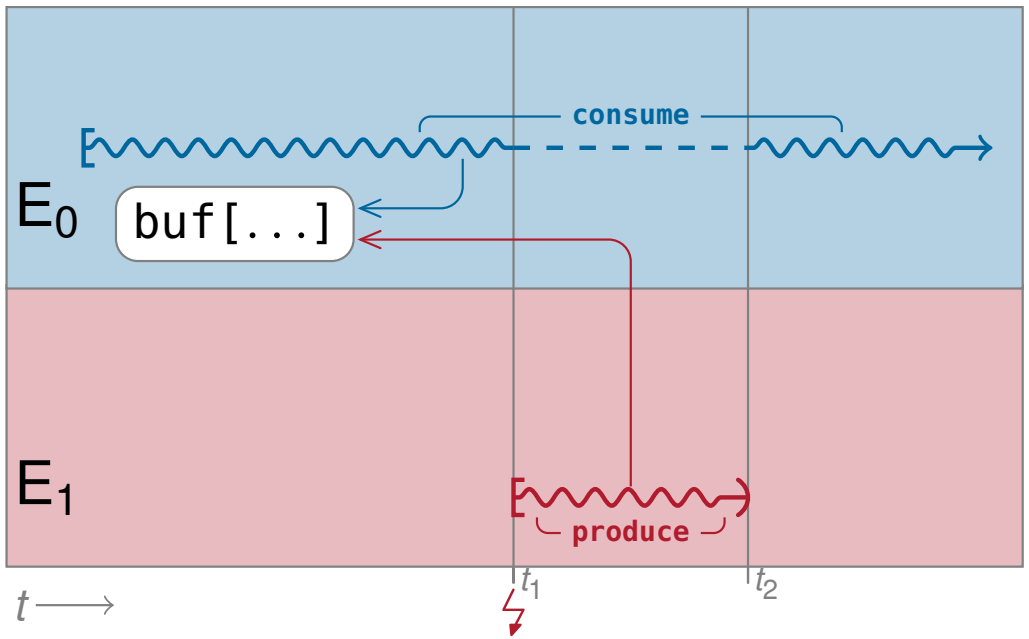
```
void produce(char data) {

      ?

}
```



Zustand liegt (logisch) auf E_0

Bounded Buffer – Konsistenzbedingungen, Annahmen

- Konsistenzbedingung
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - *entweder* `consume()` vor `produce()` *oder* `consume()` nach `produce()`
- Annahmen
  - `produce()` unterbricht `consume()`
    - alle anderen Kombinationen kommen nicht vor
  - `produce()` läuft immer durch (*run-to-completion*)



Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {    // Unterbrechungsbehandlung:
    int elements = occupied;       // Elementzaehler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zaehler erhoehen
  }
  char consume() {             // normaler Kontrollfluss:
    int elements = occupied;       // Elementzaehler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zaehler erniedrigen
    return result;                 // Ergebnis zurueckliefern
} };
```

Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

Insbesondere Zustand, auf den von beiden Seiten **schreibend** zugegriffen wird.

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nexti
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {    // Unterbrechungsbehandlung:
    int elements = occupied;       // Elementzaehler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zaehler erhoehen
  }
  char consume() {             // normaler Kontrollfluss:
    int elements = occupied;       // Elementzaehler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zaehler erniedrigen
    return result;                 // Ergebnis zurueckliefern
} };
```

Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Diese alternative Implementierung kommt ohne gemeinsam beschriebenen Zustand aus.

Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Allerdings gibt es hier jetzt Zustand, der von einer Seite gelesen und von der jeweils anderen beschrieben wird.

An genau diesen Stellen müssen wir prüfen, ob die Konsistenzbedingung gilt.

Bounded Buffer – Analyse der neuen Implementierung

- Angenommen, die Unterbrechung von `consume()` erfolgt:
  - aus der Sicht von `consume()`
    - vor dem Lesen von **`nextin`** ⇔ `consume()` nach `produce()` ✔
    - nach dem Lesen von **`nextin`** ⇔ `consume()` vor `produce()` ✔
  - aus der Sicht von `produce()`
    - vor dem Schreiben von **`nextout`** ⇔ `produce()` vor `consume()` ✔
    - nach dem Schreiben von **`nextout`** ⇔ `produce()` nach `consume()` ✔

```
char consume() {
  if (nextout == nextin) return 0;
  char result = buf[nextout];
  nextout = (nextout + 1) % SIZE;
  return result;
}
```

Konsistenzbedingung ist in jedem Fall erfüllt!

```
void produce(char data) {
  if ((nextin + 1) % SIZE == nextout) return;
  buf[nextin] = data;
  nextin = (nextin + 1) % SIZE;
}
```

Systemzeit – Implementierung aus der letzten Vorlesung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
```

```
/* Systemzeit abfragen */

  time_t time () {
    return global_time;
}
```

```
/* Unterbrechungs- *
 * behandlung      */
void timerHandler () {
  global_time++;
}
```

h8300-hms-g++ (16-Bit-Architektur)

```
time:
  mov global_time, %r0; lo
  mov global_time+2, %r1; hi
  ret
```

Problem:
Daten werden nicht atomar gelesen.

Systemzeit – Konsistenzbedingungen, Annahmen, Ansatz

- Konsistenzbedingung
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - *entweder* `time()` vor `timerHandler()` *oder umgekehrt*
- Annahmen
  - `timerHandler()` unterbricht `time()`
    - alle anderen Kombinationen kommen nicht vor
  - `timerHandler()` läuft immer durch (*run-to-completion*)
- Lösungsansatz: In `time()` optimistisch herangehen

1. lese Daten unter der Annahme nicht unterbrochen zu werden
2. überprüfe, ob Annahme zutraf – wurden wir unterbrochen?
3. falls unterbrochen, setze neu auf ab Schritt 1

Systemzeit – Neue Implementierung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
extern volatile bool interrupted;
```

```
/* Systemzeit abfragen */
time_t time () {
  time_t res;
  do {
    interrupted = false;
    res = global_time;
  } while (interrupted);
  return res;
}
```

```
/* Unterbrechungsbehandlung */
void timerHandler () {
  interrupted = true;
  global_time++;
}
```

Konsistenzbedingung ist nun in jedem Fall erfüllt!

Weiche Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt (durch Unterbrechungstransparenz)
  - Priorität wird nie verletzt
    - Kontrollflüsse der höherprioren Ebenen kommen immer durch
  - Kosten entstehen entweder gar nicht oder nur im Konfliktfall
    - gar nicht ⇝ Beispiel Bounded Buffer
    - im Konfliktfall ⇝ optimistische Verfahren, Beispiel Systemzeit (zusätzliche Kosten durch Wiederaufsetzen)
- **Nachteile**
  - Lösungen häufig sehr komplex
    - Wenn man überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen – und noch schwieriger zu verifizieren
  - Lösungen häufig sehr fragil (bezüglich Randbedingungen)
    - Kleinste Änderungen können die Konsistenzgarantie zerstören
    - Codegenerierung des Compilers ist zu beachten
  - Bei größeren Datenmengen steigen die Wiederaufsetzkosten

Weiche Synchronisation: Bewertung (Forts.)

Fazit

- Weiche Synchronisation durch Unterbrechungstransparenz ist **grundsätzlich erstrebenswert**!
- Es handelt sich bei den Algorithmen jedoch immer um **Speziallösungen** für **Spezialfälle**.
- Als allgemein verwendbares Mittel für die Sicherung **beliebiger Datenstrukturen** ist sie **nicht geeignet**.

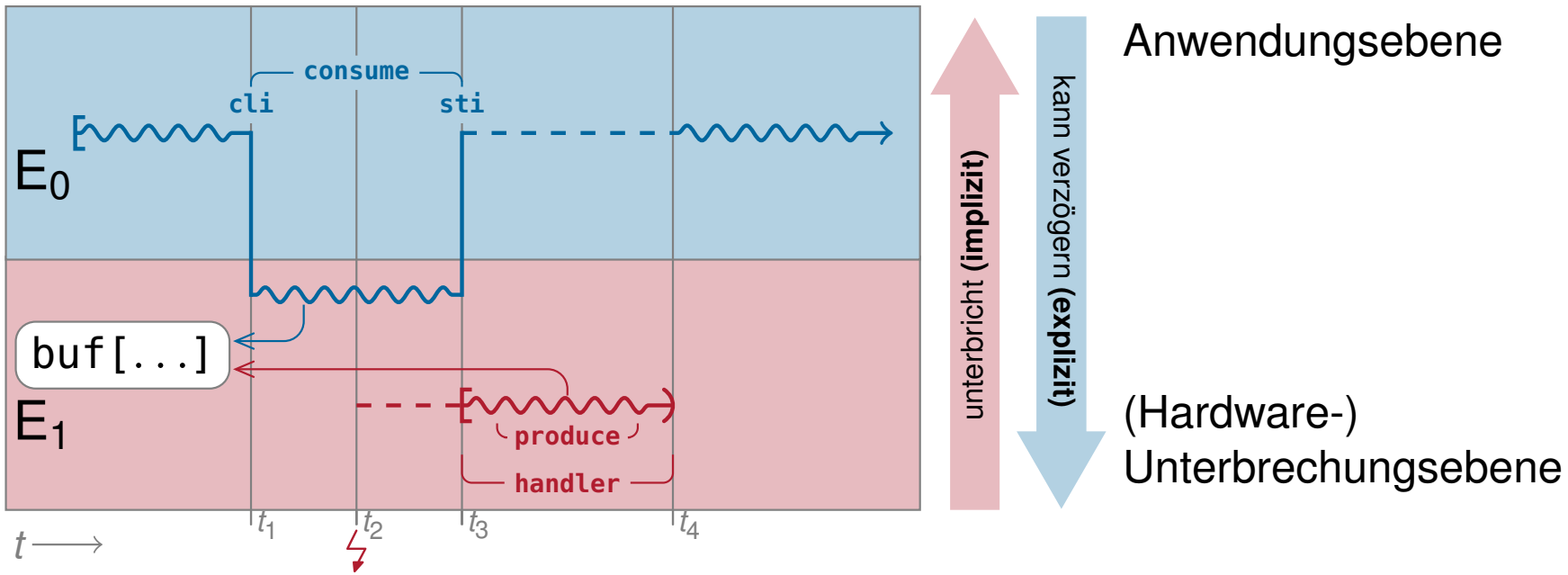
Agenda

Einleitung
Prioritätsebenenmodell
Harte Synchronisation
Weiche Synchronisation
Prolog/Epilog-Modell
- Ansatz
- Implementierung
- Bewertung

Zusammenfassung
Referenzen

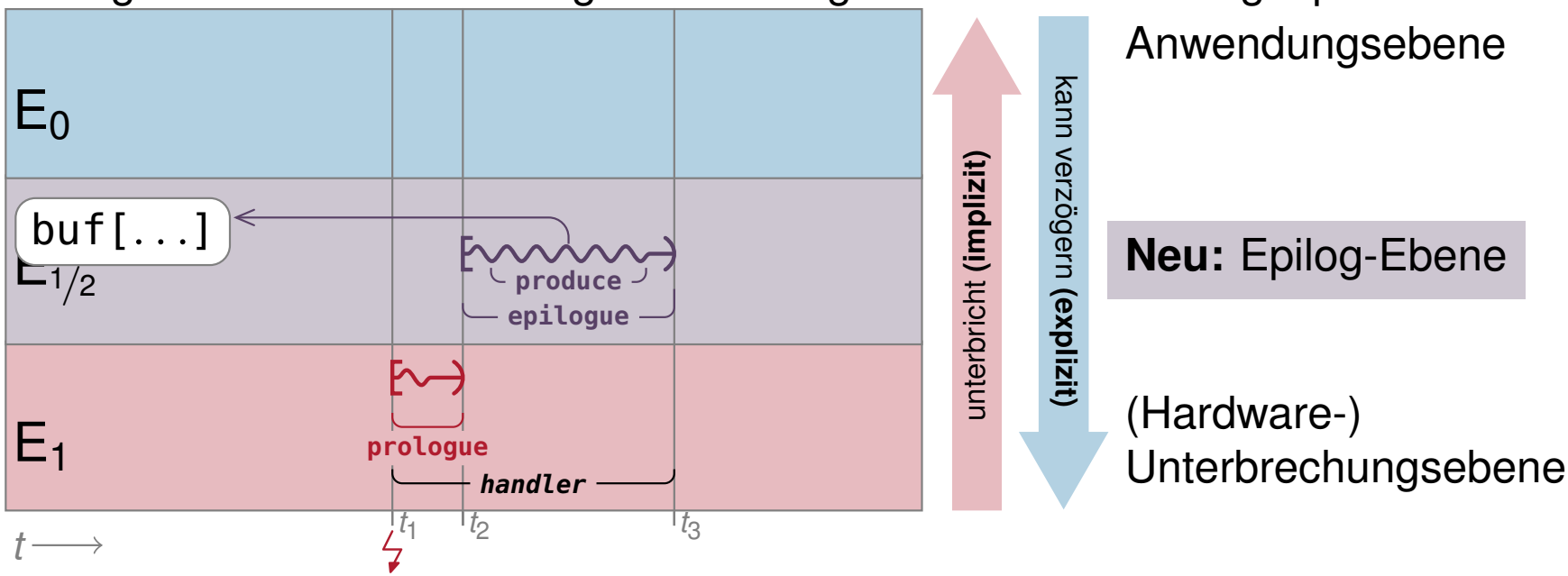
Prolog/Epilog-Modell – Motivation

- **Reprise:** Harte Synchronisation
  - einfach, korrekt, „funktioniert immer“ ✔
  - Hauptproblem ist die hohe Latenz ✘
    - Verzögerung bei Zugriff auf den Zustand aus höheren Ebenen
    - Verzögerung bei Bearbeitung des Zustands in der UB selbst
  - letztlich dadurch verursacht, dass der Zustand (logisch) auf der/einer Hardwareunterbrechungsebene $E_{1\ldots n}$ liegt.



Prolog/Epilog-Modell – Ansatz

- **Ansatz:** Latenzverbergung durch zusätzliche Ebene
  - Wir fügen eine weitere *logische* Ebene ein: $E_{1/2}$
    - $E_{1/2}$ liegt zwischen der Anwendungsebene E_0 und den UB-Ebenen $E_{1\ldots n}$
  - Unterbrechungsbehandlung wird zweigeteilt in Prolog und Epilog
    - Prolog arbeitet auf Unterbrechungsebene $E_{1\ldots n}$
    - Epilog arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (Epilogebene)
  - Zustand liegt (so weit wie möglich) auf der Epilogebene
    - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



Prolog/Epilog-Modell – Ansatz (Forts.)

- Unterbrechungsbehandlungsroutinen werden zweigeteilt
  - beginnen im **Prolog** (immer)
  - werden fortgesetzt im **Epilog** (bei Bedarf)
- **Prolog** ($\rightsquigarrow$ Hardwareunterbrechung)
  - läuft auf Hardwareunterbrechungsebene
    - hat damit Priorität über Anwendungsebene und Epilogebene
  - ist **kurz**, fasst wenig oder gar keinen Zustand an
    - Üblicherweise wird nur der Hardware-Zustand gesichert und bestätigt
    - Unterbrechungen bleiben nur kurz gesperrt ($\rightsquigarrow$ Latenzminimierung)
    - kann bei Bedarf einen Epilog für die weitere Verarbeitung anfordern
- **Epilog** ($\rightsquigarrow$ Softwareunterbrechung)
  - läuft auf Epilogebene $E_{1/2}$ (zusätzliche Kontrollflussebene)
    - Ausführung erfolgt verzögert zum Prolog
    - erledigt die eigentliche Arbeit ($\rightsquigarrow$ Latenzverbergung)
  - hat Zugriff auf größten Teil des Zustands
    - Zustand wird auf Epilogebene synchronisiert

Prolog/Epilog-Modell – Epilogebene

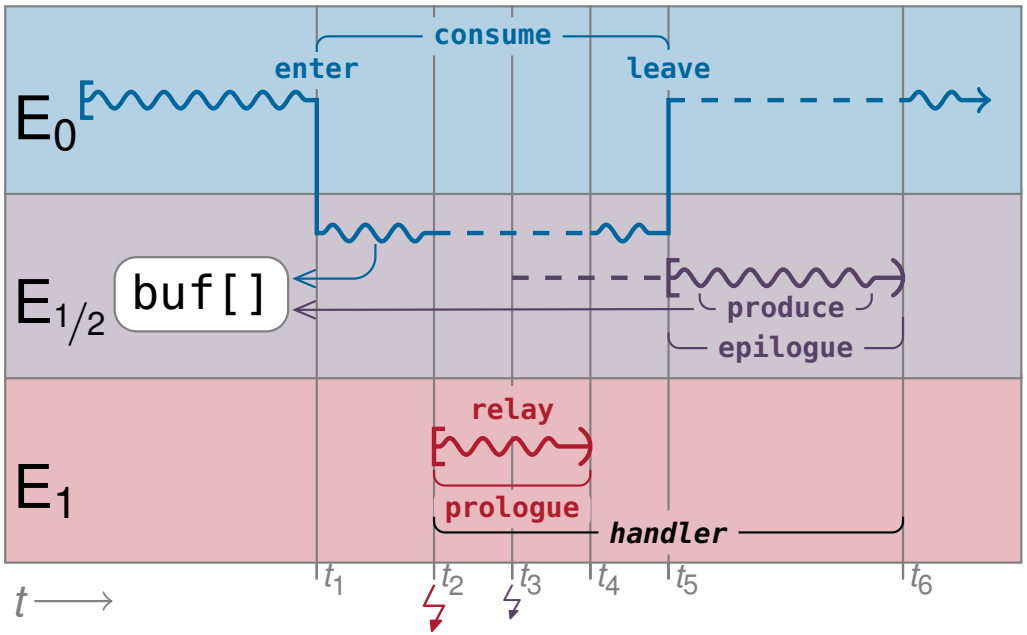
- Die Epilogebene wird (ganz oder teilweise) in Software implementiert
  - trotzdem handelt es sich um eine ganz normale Prioritätsebene des Ebenenmodells
  - es müssen daher auch dieselben Gesetzmäßigkeiten gelten

- Es gilt: Kontrollflüsse auf der Epilogebene $E_{1/2}$ werden
  1. jederzeit unterbrochen durch Kontrollflüsse der Ebenen $E_{1\ldots n}$
     $\rightsquigarrow$ Prologe (Unterbrechungen) haben Priorität über Epiloge
  2. nie unterbrochen durch Kontrollflüsse der Ebene E_0
     $\rightsquigarrow$ Epiloge haben Priorität über Anwendungskontrollflüsse
  3. sequentialisiert mit anderen Kontrollflüssen von $E_{1/2}$
     $\rightsquigarrow$ Anhängige Epiloge werden nacheinander abgearbeitet.
     $\rightsquigarrow$ Bei Rückkehr zur Anwendungsebene sind alle Epiloge abgearbeitet.

Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebene zu betreten: `enter()`
     - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebene zu verlassen: `leave()`
     - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: `relay()`
     - entspricht dem Hochziehen der IRQ-Leitung beim PIC

Prolog/Epilog-Modell – Ablaufbeispiel



E_1-Unterbrechungen werden nie gesperrt.

Aktivierungslatenz der Unterbrechungsbehandlung ist minimal.

- t_1 Anwendungskontrollfluss betritt Epilogebene $E_{1/2}$ (`enter()`).
- t_2 Unterbrechung ↯ auf Ebene E_1 wird signalisiert ⇝ Prolog wird ausgeführt.
- t_3 Prolog fordert Epilog für die nachgeordnete Bearbeitung an (`relay()` ↯).
- t_4 Prolog terminiert, unterbrochener $E_{1/2}$-Kontrollfluss läuft weiter.
- t_5 Anwendungskontrollfluss verlässt die Epilogebene $E_{1/2}$ (`leave()`)
  ⇝ zwischenzeitlich aufgelaufene Epiloge werden nun abgearbeitet.
- t_6 Epilog terminiert, Anwendungskontrollfluss fährt auf E_0 fort.

Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebene zu betreten: `enter()`
     - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebene zu verlassen: `leave()`
     - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: `relay()`
     - entspricht dem Hochziehen der IRQ-Leitung beim PIC
- Außerdem Mechanismen, um
  4. anhängige Epiloge zu „merken“: `queue` (z. B.)
     - entspricht dem IRR (Interrupt-Request-Register) beim PIC
  5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
     - entspricht bei der harten Synchronisation dem Protokoll zwischen CPU und PIC

Dieser Punkt muss etwas genauer betrachtet werden!

Prolog/Epilog-Modell – Implementierung

5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
   - entspricht bei der harten Synchronisation
     dem Protokoll zwischen CPU und PIC

Wann müssen anhängige Epiloge abgearbeitet werden?

Immer unmittelbar, bevor die CPU auf E_0 zurückkehrt!

1. bei explizitem Verlassen der Epilogebene mit `leave()`
   - während der Anwendungskrontrollfluss auf $E_{1/2}$ gearbeitet hat könnten Epiloge aufgelaufen sein (↩ Sequentialisierung).
2. nach Abarbeitung des letzten Epilogs
   - während der Epilogabarbeitung könnten weitere Epiloge aufgelaufen sein (↩ Sequentialisierung).
3. wenn der **letzte** Unterbrechungsbehandler terminiert
   - während der Abarbeitung von $E_{1\ldots n}$-Kontrollflüssen könnten Epiloge aufgelaufen sein (↩ Priorisierung).

Prolog/Epilog-Modell – Implementierung

- Implementierungsvarianten
  - rein softwarebasiert (⇝ **Übung**)
  - mit Hardwareunterstützung durch einen AST (⇝ [1, 2])
- Ein AST (*asynchronous system trap*) ist eine Unterbrechung, die (nur) durch Software angefordert werden kann.
  - z. B. durch Setzen eines Bits in einem bestimmten Register
  - ansonsten technisch vergleichbar mit einer Hardware-Unterbrechung
    - AST wird (im Gegensatz zu Traps/Exceptions) asynchron abgearbeitet
    - AST läuft auf eigener Unterbrechungsebene zwischen der Anwendungsebene und den Hardware-UBs (↪ unsere $E_{1/2}$)
    - Gesetzmäßigkeiten des Ebenenmodels gelten (AST-Ausführung ist verzögerbar, wird automatisch aktiviert, . . .)
- Sicherstellung der Epilogabarbeitung wird damit sehr einfach!
  - Abarbeitung der Epiloge erfolgt im AST
    ⇝ und damit automatisch, bevor die CPU auf E_0 zurückkehrt
  - bleibt nur noch die Verwaltung der anhängigen Epiloge

Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E_1 konfiguriert ($\Leftrightarrow$ unsere $E_{1/2}$)
  - Geräteunterbrechungen laufen auf $E_{2...n}$

```
void enter() {
  CPU::setIRQL(1);          // betrete E1, verzoegere AST
}
void leave() {
  CPU::setIRQL(0);          // erlaube AST (anhaengiger
}                           // AST wuerde jetzt abgearbeitet)
void relay(<Epilog>) {
  <haenge Epilog an queue an>
  CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
  while(<Epilog in queue>) {
    <entferne Epilog aus queue>
    <arbeite Epilog ab>
} }
```

Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E_1 konfiguriert ($\Leftrightarrow$ unsere $E_{1/2}$)
  - Geräteunterbrechungen laufen auf $E_{2\ldots n}$

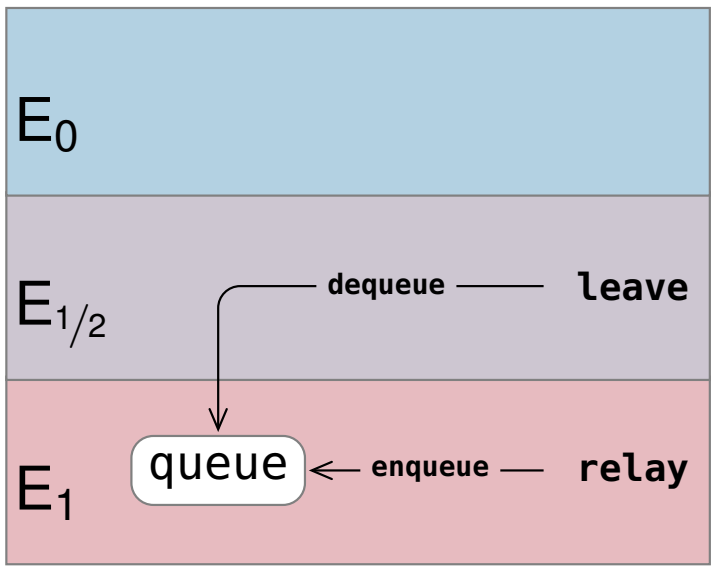
```
void enter() {
  CPU::setIRQL(1);          // b
}
void leave() {
  CPU::setIRQL(0);          // e
}                           // A
void relay(<Epilog>) {
  <haenge Epilog an queue an>
  CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
  while(<Epilog in queue>) {
    <entferne Epilog aus queue>
    <arbeite Epilog ab>
} }
```

Bietet die Hardware (wie z. B. IA-32) kein AST-Konzept, so kann man dieses in Software nachbilden.

Näheres dazu in der Übung.

Prolog/Epilog-Modell – Ziel erreicht?

- Kernzustand kann jetzt auf Epilogebene verwaltet und synchronisiert werden.
  - Hardware-UBs müssen nicht (mehr) gesperrt werden!
- Ein Problem bleibt noch: Die Epilog-Warteschlange
  - Zugriff erfolgt aus Prologen und der Epilogebene
    - muss also entweder hart synchronisiert werden (im Bild)
    - oder man sucht eine Speziallösung mit weicher Synchronisation



Harte Synchronisation erscheint hier akzeptabel, da die Sperrzeit ($\Leftrightarrow$ Ausführungszeit von `dequeue()`) kurz und deterministisch ist.

Eine Lösung mit weicher Synchronisation (z. B. [3]) wäre natürlich schöner!

Prolog/Epilog-Modell: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt
    (durch Synchronisation auf Epilogebene)
  - Programmiermodell entspricht dem (einfach verständlichen) Modell der harten Synchronisation
  - Auch komplexer Zustand kann synchronisiert werden
    - ohne das dabei Unterbrechungsanforderungen verloren gehen
    - ermöglicht es, den gesamten BS-Kern auf Epilogebene zu schützen
- **Nachteile**
  - Zusätzliche Ebene führt zu zusätzlichem Overhead
    - Epilogaktivierung könnte länger dauern als direkte Behandlung
    - Komplexität für den BS-Entwickler wird erhöht
  - Unterbrechungsperren lassen sich nicht vollständig vermeiden
    - Gemeinsamer Zustand von Pro- und Epilog muss weiter hart oder weich synchronisiert werden

Prolog/Epilog-Modell: Bewertung (Forts.)

Fazit

- Das Prolog/Epilog-Modell ist ein **guter Kompromiss** für die Synchronisation des Kernzustands.
- Es ist auch für die Konsistenzsicherung **komplexer Datenstrukturen geeignet**

Agenda

Einleitung
Prioritätsebenenmodell
Harte Synchronisation
Weiche Synchronisation
Prolog/Epilog-Modell
Zusammenfassung
Referenzen

Zusammenfassung: Unterbrechungssynchronisation

- Konsistenzsicherung im BS-Kern
  - muss anders erfolgen als zwischen Prozessen – einseitig
  - Kontrollflüsse arbeiten auf verschiedenen Prioritätsebenen
- Maßnahmen zur Konsistenzsicherung
  - harte Synchronisation (durch Unterbrechungssperren)
    - einfach, jedoch negative Auswirkungen auf Latenz
    - Unterbrechungsanforderungen können verloren gehen
  - weiche Synchronisation (durch Unterbrechungstransparenz)
    - gut und effizient, jedoch nur in Spezialfällen möglich
    - Implementierung kann sehr komplex werden
  - Prolog/Epilog-basierte Synchronisation (Zweiteilung der Unterbrechungsbehandlung)
    - guter Kompromiss
    - Stand der Technik in heutigen Betriebssystemen

Nachtrag: Mehrkernsysteme

Beachte: Unterbrechungsbehandlung $\neq$ Parallelität

- Techniken funktionieren (so) nur bei echter Unterbrechungssemantik: A und UB werden auf **demselben** Prozessor ausgeführt
- Wird die UB „echt parallel” (auf einem weiteren Prozessor) ausgeführt, kommt es zu Problemen
  - Annahmen des Prioritätsebenenmodells gelten nicht mehr! (Sequentialisierung, Priorisierung, *run-to-completion*)
  - Asymmetrie (UB unterbricht A) ist nicht länger gegeben (weiche Synchronisation wird dadurch viel schwieriger)
- Zusätzlich erforderlich: **Interprozessor-Synchronisation**
  - „hart” $\mapsto$ zweiseitig blockierend, z. B. mit *Spin-Locks* $\rightsquigarrow$ Übung
  - „weich” $\mapsto$ algorithmisch nichtblockierend (**schwer!**) $\rightsquigarrow$ [CS]

Referenzen

Digital Equipment Corporation. *VAX-11 Architecture Reference Manual*. Document Number EK-VAXAR-RM-001. Digital Equipment Corporation. Maynard, MA, USA: Digital Press, Mai 1982.

Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels u. a. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Mai 1989. ISBN: 0-201-06196-1.

Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk u. a. „On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System“. In: *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '00)* (Newport Beach, CA, USA). IEEE Computer Society Press, März 2000, S. 270–277. DOI: 10.1109/ISORC.2000.839540.

Wolfgang Schröder-Preikschat. *Concurrent Systems*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_CS.

Wolfgang Schröder-Preikschat. *Systemprogrammierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_SP.

Betriebssysteme (BS)

VL 7 – IA-32: Das Programmiermodell der Intel-Architektur

Volkmar Sieh / **Daniel Lohmann**

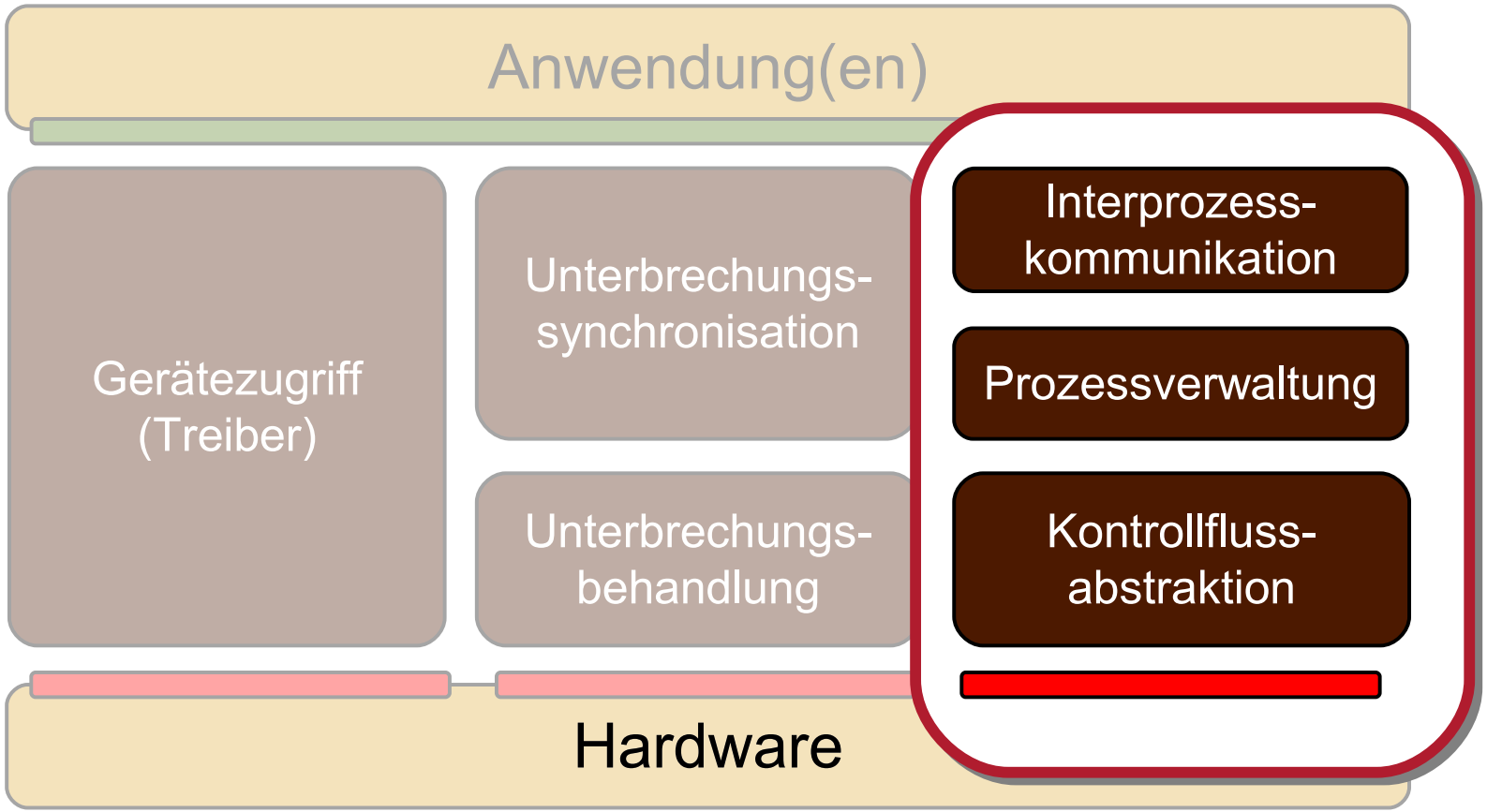
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 6. Dezember 2023

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Agenda

Einordnung
Urvater: Der 8086
Die 32-Bit Intel-Architektur
Protected Mode
Multitasking
Zusammenfassung

Historie der Intel x86-Prozessoren

- 1978: **8086** *der Urvater des PC Prozessors*
- 1982: **80286** *Einführung des Protected Mode*
  - segmentbasierter Speicherschutz
- 1985: **80386** *erster IA-32 Prozessor*
  - seitenbasierter virtueller Speicher
  - Protected Mode
- 1989: **80486** *integrierte FPU, RISC Ansätze*
- 1993: **Pentium** *P5-Architektur*
  - superskalar, 64-Bit Datenbus
  - SMM, MMX, APIC, Dualprozessor-fähig

Historie der Intel x86-Prozessoren (Forts.)

- 1995: **Pentium Pro** *P6-Architektur*
  - 36-Bit Adressbus (PAE)
  - Level 2 Cache on Chip, RISC-artige Mikroinstruktionen
- 1997: **Pentium II** *Pentium Pro + MMX*
  - Level 2 Cache wieder extern, dafür bessere 16-Bit Performance
- 1999: **Pentium III** *SSE, Pentium M (2003)*
- 2000: **P4** *Netburst-Architektur*
  - SSE2, optimiert für hohe Taktzahlen (angedacht bis zu 10 GHz)
- 2004: **P4 Prescott** *Erweiterte Netburst Architektur*
  - Hyperthreading, Vanderpool, EM64T, 31-stufige Pipeline!

Historie der Intel x86-Prozessoren (Forts.)

- 2005: **Core** *Ende der Netburst Architektur*
  - geringerer Takt, weniger Strom, aber bessere Performance!
  - Architektur basiert auf P6-Architektur, kein Hyperthreading
- 2006: **Core 2** *Dual Core, Quad Core, 64 Bit*
- 2008: **Atom** *extrem stromsparend*
  - Architektur (wieder) CISC-lastiger, Ähnlichkeiten mit 486/Pentium
- 2009: **Core i7** *Nehalem-Architektur*
  Sandy-Bridge-Architektur (2011)
  Haswell-Architektur (2013)
  Skylake-Architektur (2017)
  - Hyperthreading, Octa Core, Quick Path Interconnect, AVX
  - “Power Control Unit” (PCU) passt Takt der TDP an
- 2012: **Xeon Phi** *Larrabee-Architektur*
  - P54C Manycore (62 cores), SIMD Instruktionen

Agenda

Einordnung

Urvater: Der 8086

- Programmiermodell
- Speichermodell

Die 32-Bit Intel-Architektur

Protected Mode

Multitasking

Zusammenfassung

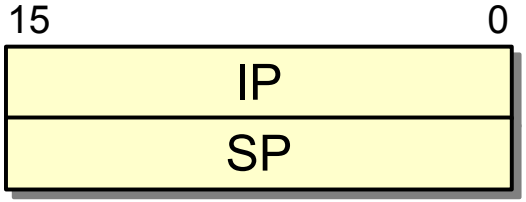
8086: Programmiermodell

- 16-Bit Architektur, little-endian
- 20-Bit Adressbus, d.h. maximal 1 MiB Hauptspeicher
- wenige Register
  - (jedenfalls aus heutiger Sicht)
- 123 Befehle
  - kein orthogonaler Befehlssatz
- Befehlslängen von 1 bis 4 Byte
- segmentierter Speicher
- **noch immer aktuell**
  - obwohl von 1978 noch heute von jeder IA-32 CPU unterstützt
    - *Real Mode*, *Virtual 8086 Mode*

Aufwärtskompatibilität wird bei Intel groß geschrieben

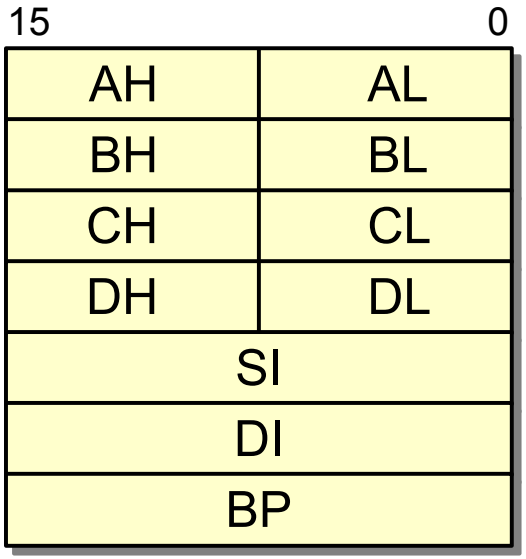
8086: Registersatz

Befehls- und Stapelzeiger

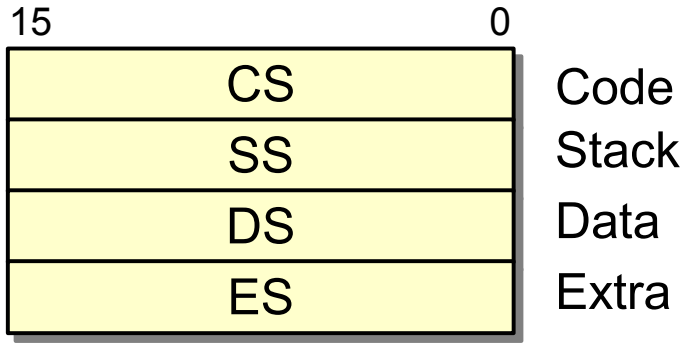


Flag Register

Vielzweckregister

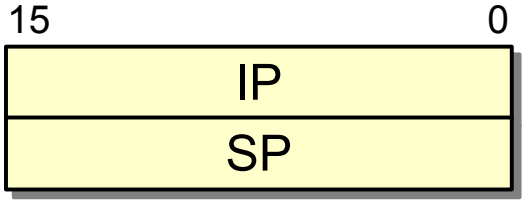


Segmentregister

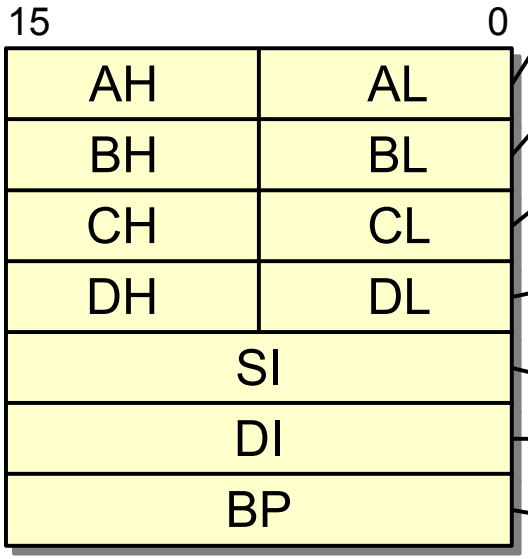


8086: Registersatz

Befehls- und Stapelzeiger



Vielzweckregister



AX: Accumulator Register
- arithmetisch-logische Operationen
- I/O
- kürzester Maschinencode

BX: Base Address Register

CX: Count Register
- für LOOP Befehl
- für *String* Operationen mit REP
- für Bit *Shift* und *Rotate*

DX: Data Register
- DX:AX sind 32 Bit für MUL/DIV
- Portnummer für IN und OUT

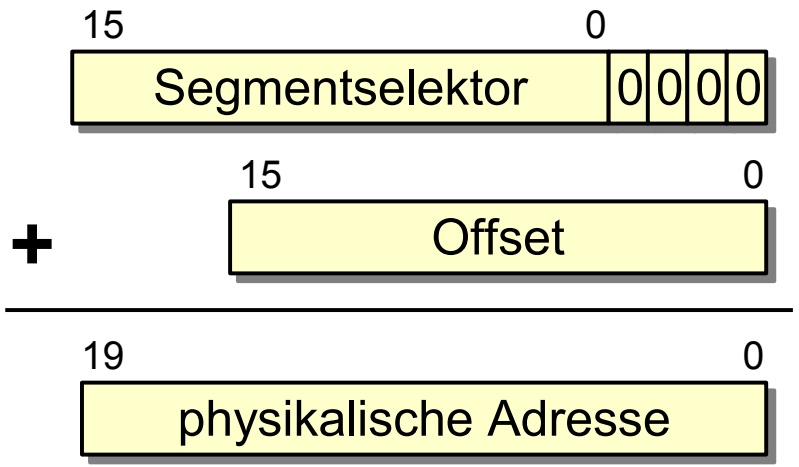
SI, DI: Index Register
- für Array-Zugriffe (Displacement)

BP: Base Pointer

Jedes „Vielzweckregister“ erfüllt seinen speziellen Zweck

8086: Segmentierter Speicher

- logische Adressen bestehen beim 8086 aus
  - Segmentselektor (i.d.R. der Inhalt eines Segmentregisters)
  - Offset (i.d.R. aus einem Vielzweckregister oder dem Befehl)
- Berechnung der physikalischen Adresse:



die 16 Bit Konkurrenz konnte i.d.R. nur 64KB adressieren

8086: Segmentierter Speicher

- logische Adressen bestehen beim 8086 aus
  - Segmentselektor (i.d.R. der Inhalt eines Segmentregisters)
  - Offset (i.d.R. aus einem Vielzweckregister oder dem Befehl)
- Ber

> *„640K ought to be enough for anybody“*
>
> angeblich ein Zitat von Bill Gates, 1981

physikalische Adresse

die 16 Bit Konkurrenz konnte i.d.R. nur 64KB adressieren

8086: Speichermodelle

- Programme können Adressen unterschiedlich bilden. Das Ergebnis waren unterschiedliche Speichermodelle:
  - **Tiny**
    - Code-, Daten- und Stacksegment sind identisch: 64K insgesamt
  - **Small**
    - Trennung des Codes von Daten und Stack: 64K + 64K
  - **Medium**
    - 32(20) Bit Zeiger für Code, Daten- und Stapelseg. aber fest (64K)
  - **Compact**
    - Festes Code Segment (64K), 32(20) Bit Zeiger für Daten und Stack
  - **Large**
    - „far“ Zeiger für alles: 1MB komplett nutzbar
  - **Huge**
    - wie „Large“, aber mit normalisierten Zeigern

8086: Fazit

- Urvater der PC-Prozessoren
  - bildete den Kern der ersten PCs
  - noch heute sind IA32-Prozessoren kompatibel
- Segmentregister brachten Vorteile
  - trotz 16-Bit-Architektur 1 MB Speicher
  - Trennung von logischen Modulen im Hauptspeicher
- Programm- und Übersetzerentwicklung ist aber vergleichsweise schwierig
  - verschiedene Speichermodelle
  - nicht orthogonaler Befehlssatz

Agenda

Einordnung

Urvater: Der 8086

Die 32-Bit Intel-Architektur

- Erweiterungen
- A20-Gate

Protected Mode

Multitasking

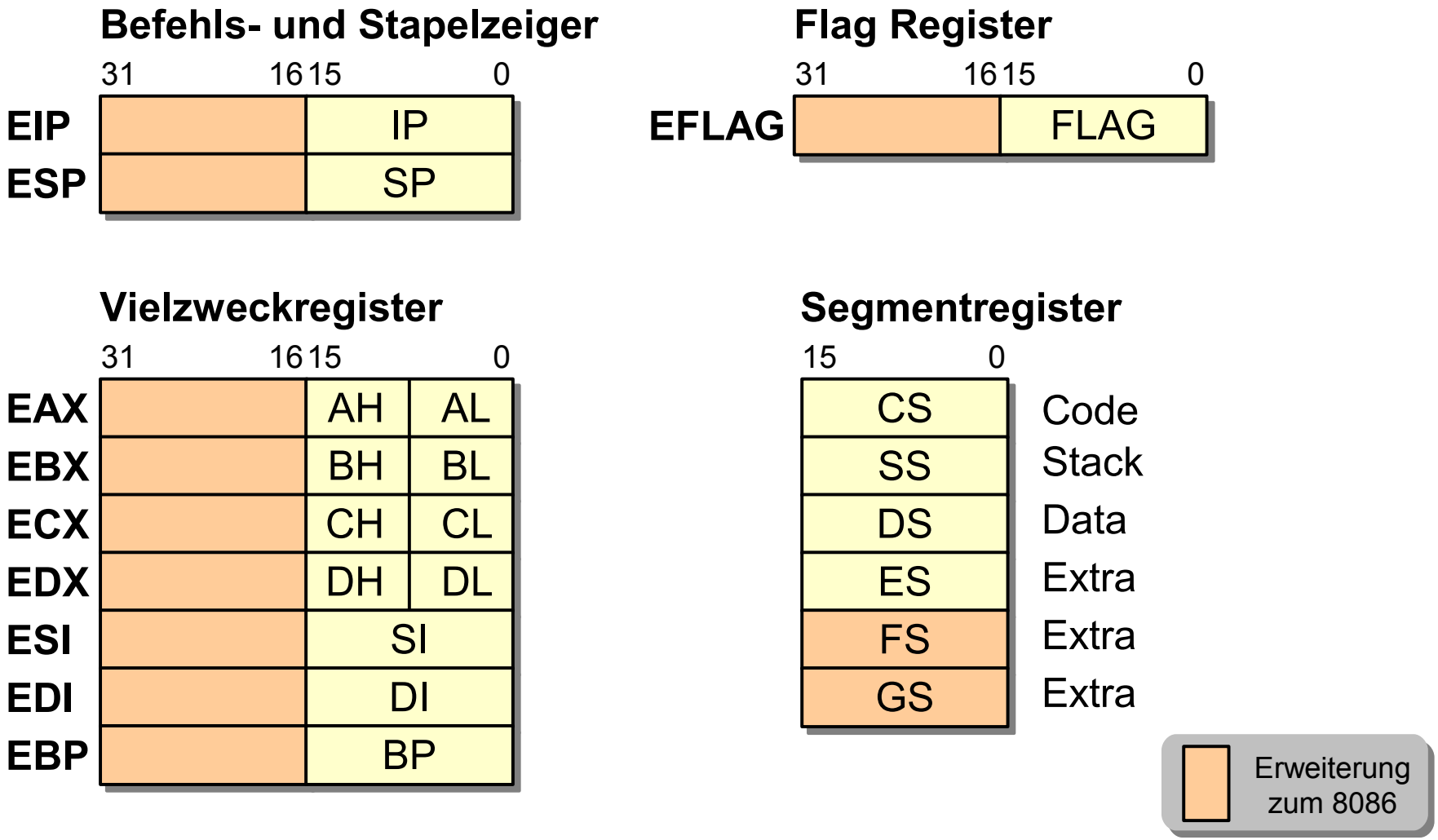
Zusammenfassung

IA-32 – die 32 Bit Intel Architektur

- die erste IA-32 CPU war der **80386**
  - wobei der Begriff „IA-32“ erst sehr viel später eingeführt wurde
- 32 Bit Technologie: Register, Daten- und Adressbus
  - ab Pentium Pro: 64 Bit Daten und 36 Bit Adressbus
- zusätzliche Register
- komplexe Schutz- und Multitaskingunterstützung
  - *Protected Mode*
  - ursprünglich schon mit dem 80286 (16-Bit) eingeführt
- Kompatibilität
  - mit älteren Betriebssystemen durch den *Real Mode*
  - mit älteren Anwendungen durch den *Virtual 8086 Mode*
- segmentbasiertes Programmiermodell
- seitenbasierte MMU

80386: Registersatz (Erweiterungen)

- erweiterte Register heißen aus Kompatibilitätsgründen E...



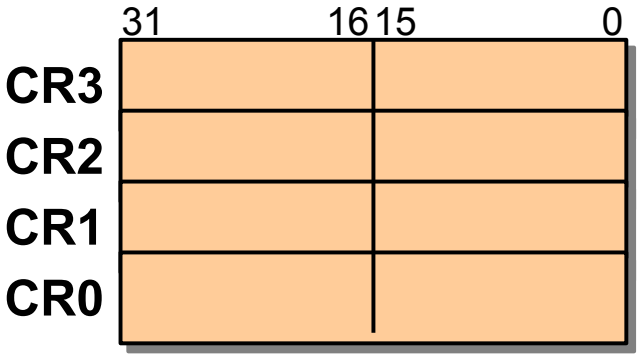
80386: Registersatz (neue Register)

Speicherverwaltungsregister

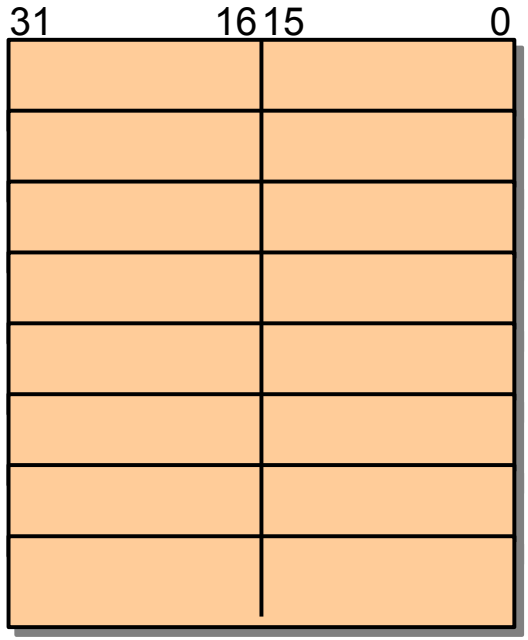
	15 – 0	31 – 0	19 – 0
TR	TSS-Sel.	TSS-Basisadresse	TSS-Limit
LDTR	LDT-Sel.	LDT-Basisadresse	LDT-Limit
IDTR		IDT-Basisadresse	IDT-Limit
GDTR		GDT-Basisadresse	GDT-Limit

Erläuterungen folgen ...

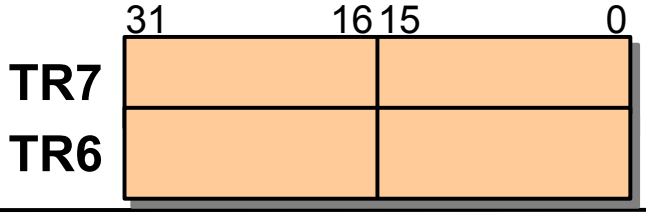
Steuerregister



Debugregister

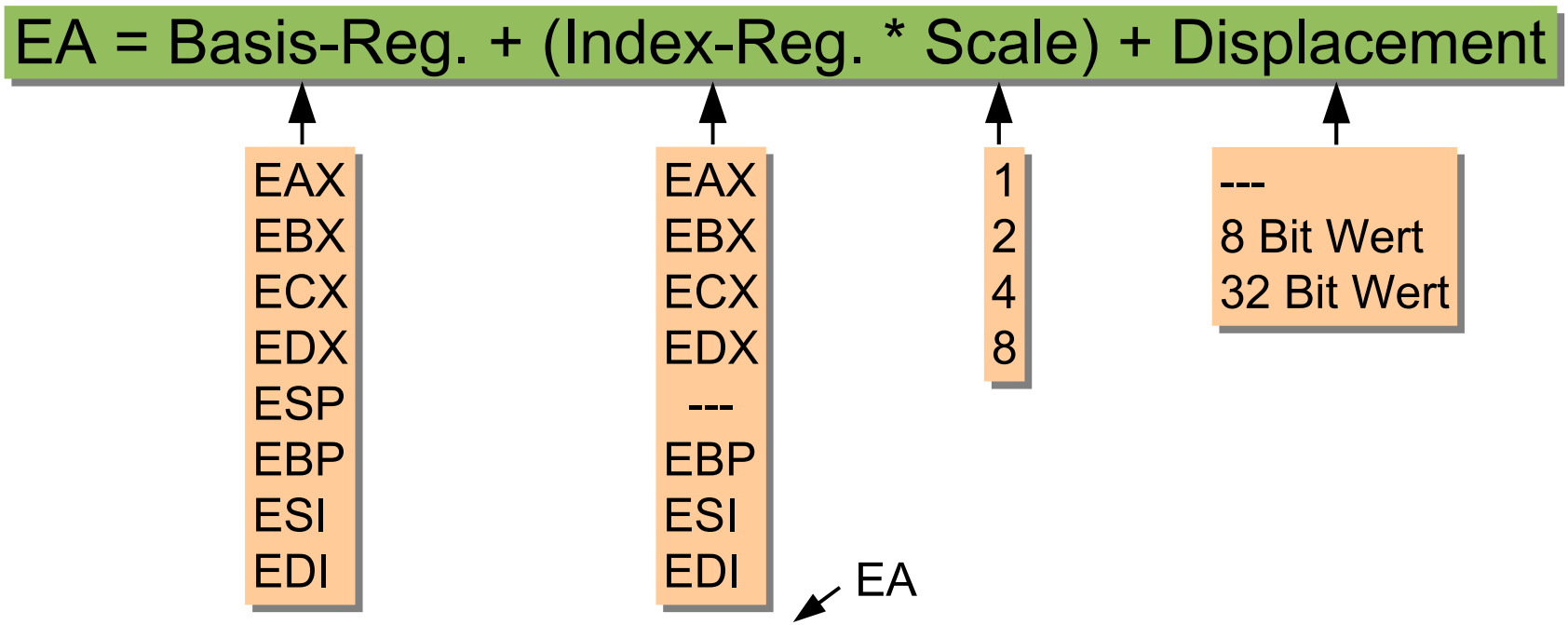


Testregister



IA-32: Adressierungsarten

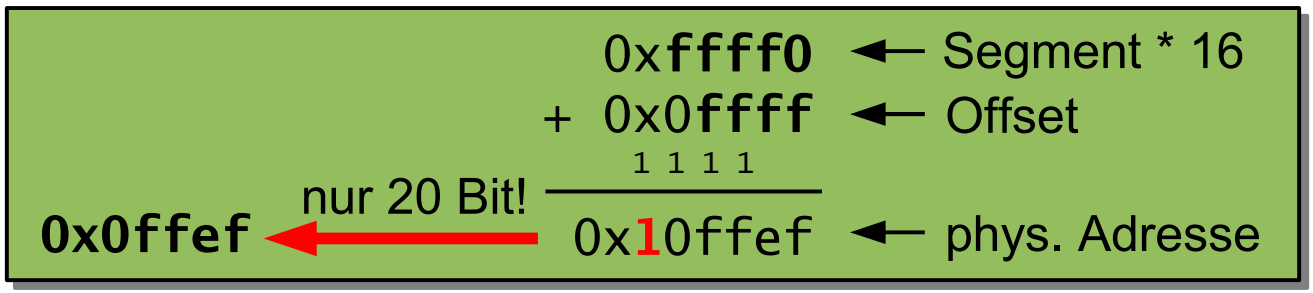
- Effektive Adressen (EA) werden nach einem einfachen Schema gebildet
  - alle Vielzweckregister können dabei gleichwertig verwendet werden



- Beispiel: MOV EAX, **Feld[ESI * 4]**
  - Lesen aus Feld mit 4 Byte großen Elementen und ESI als Index

Das A20-Gate

- ... ist ein Relikt aus der Zeit der 80286 Systeme (IBM AT)
  - beim IBM XT (8086) konnte es bei der Adressberechnung zu einem Überlauf kommen. Im Maximalfall:



  - MS-DOS (und andere Systeme) verwenden diesen „Trick“.
  - Aus Kompatibilitätsgründen wurde im IBM AT die A20-Leitung über das „A20 Gate“ (Register im **Tastaturcontroller**) maskiert.
    - A20 muss explizit freigeschaltet werden, um Speicher > 1 MiB zu addressieren
  - Ab dem 486 hat Intel das A20-Gate in die CPU integriert!

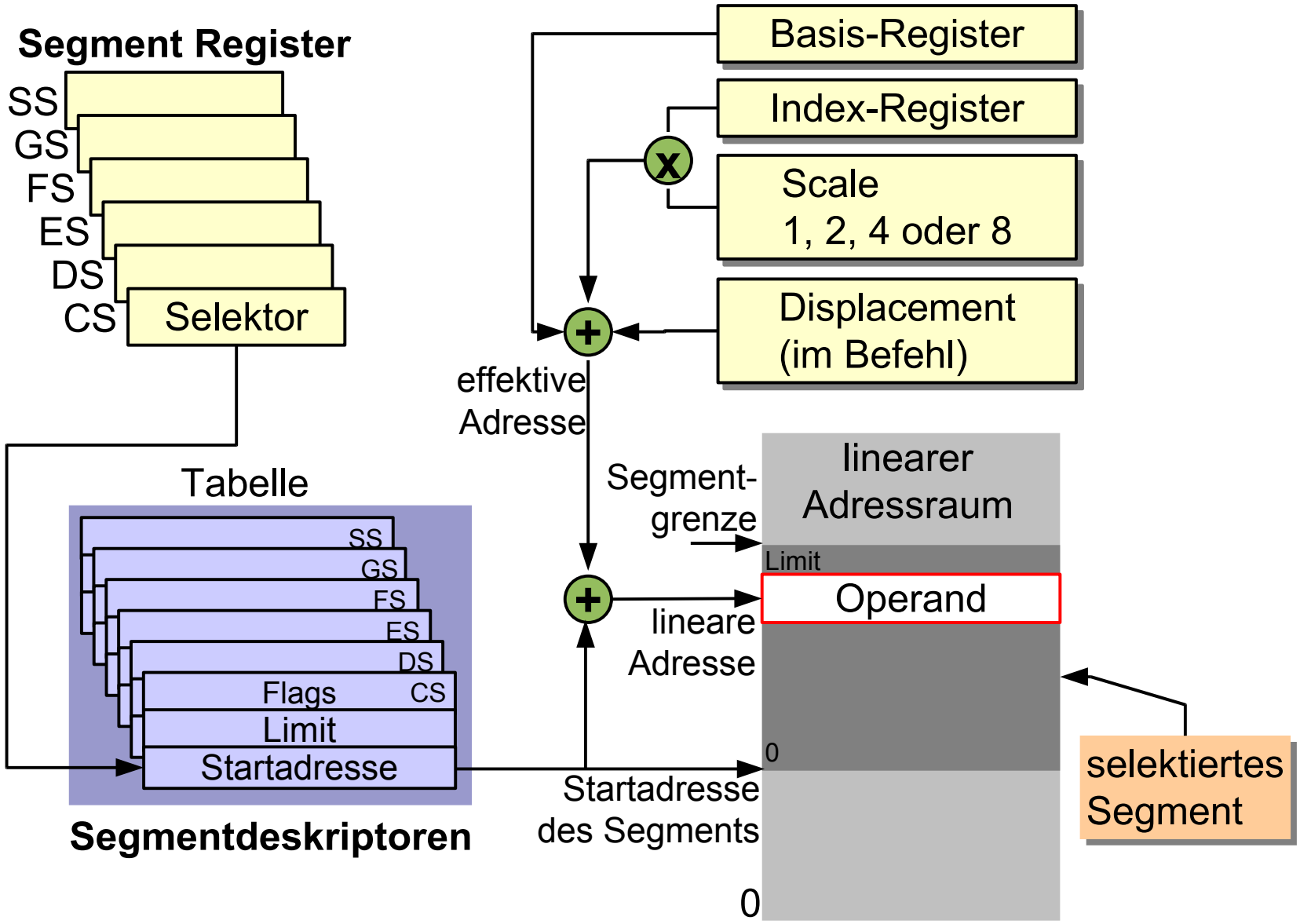
Agenda

- Einordnung
- Urvater: Der 8086
- Die 32-Bit Intel-Architektur
- Protected Mode
  - Segmente
  - MMU
  - Schutz
- Multitasking
- Zusammenfassung

IA-32: *Protected Mode* – Segmente

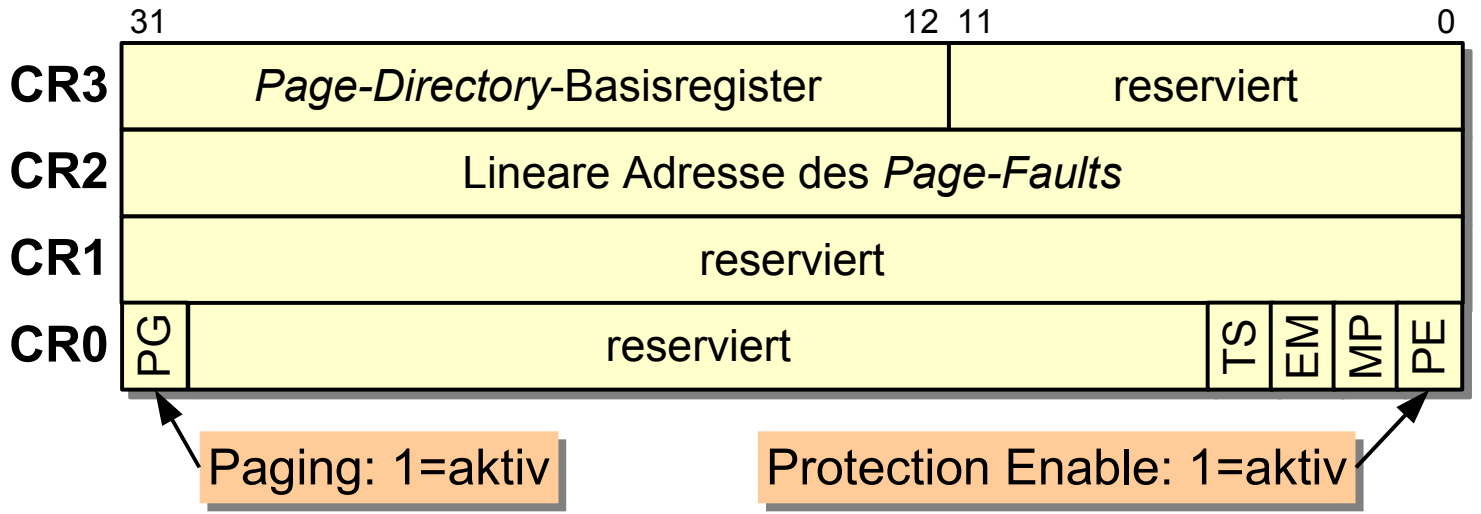
- ein Programm (in Ausführung) besteht aus mehreren Speichersegmenten
  - mindestens CODE, DATEN und STACK
  - Segmentselektoren beschreiben (indirekt) Adresse und Länge
- „Lineare Adresse“ ist Segmentstartadresse + EA
  - Segmente dürfen sich im linearen Adressraum überlappen, z.B. dürfen die Segmentstartadressen bei 0 liegen. Dadurch wird ein „flacher“ Adressraum nachgebildet.
  - „Lineare Adresse“ entspricht der physikalischen Adresse, falls die Paging Unit nicht eingeschaltet ist.

IA-32: *Protected Mode* – Segmente

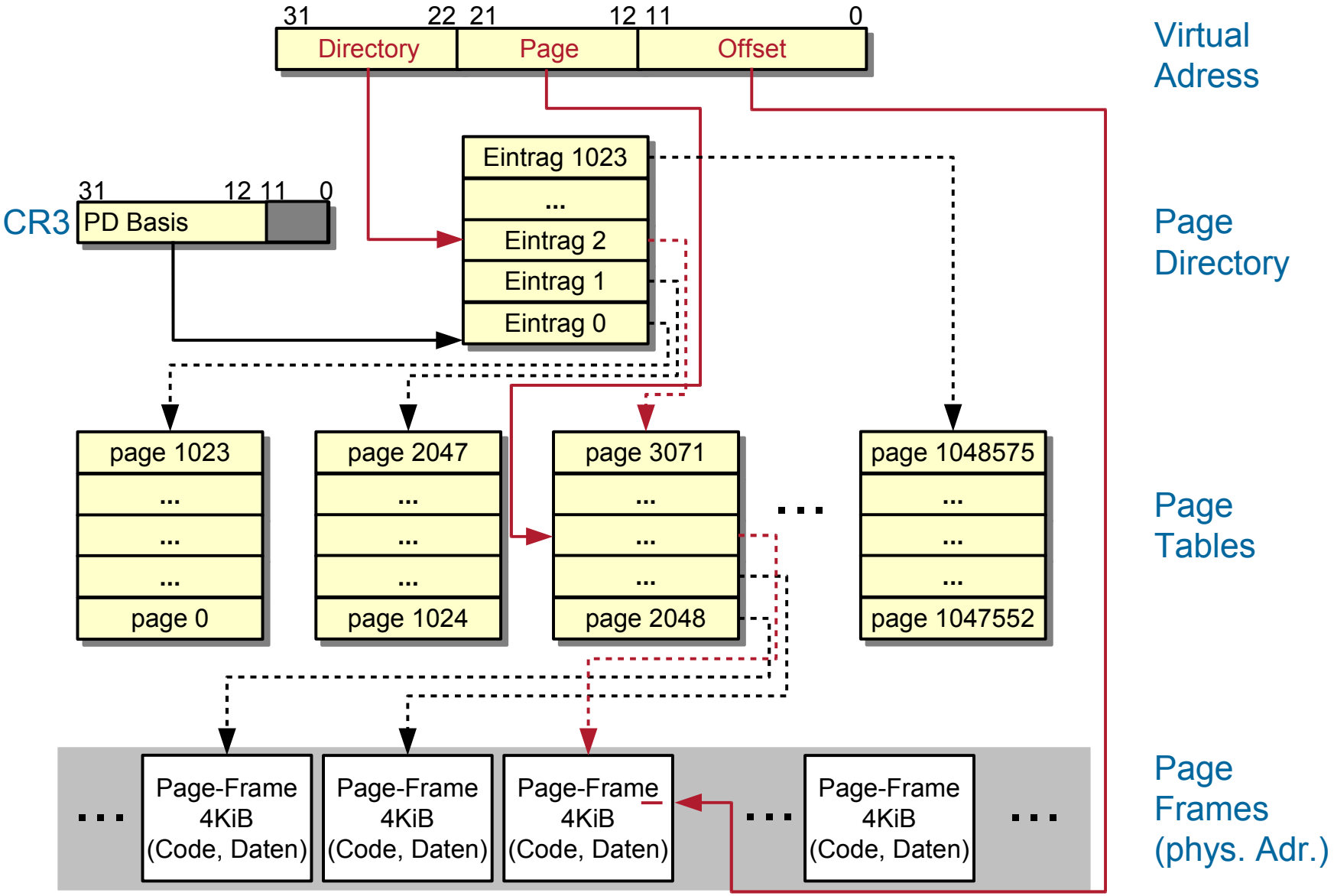


IA-32: Seitenbasierte MMU (1)

- Ein- und Auslagerung von Speicher (zwecks virtuellem Speicher) ist bei Segmentierung aufwändig. Daher bieten viele andere CPUs lediglich eine seitenbasierte Speicherverwaltung.
- ab dem 80386 kann eine ***Paging Unit*** (PU) **optional** hinzugeschaltet werden.
- die wichtigsten Verwaltungsinformationen stehen in den CRx Steuerregistern:



IA-32: Seitenbasierte MMU (2)



IA-32: TLB

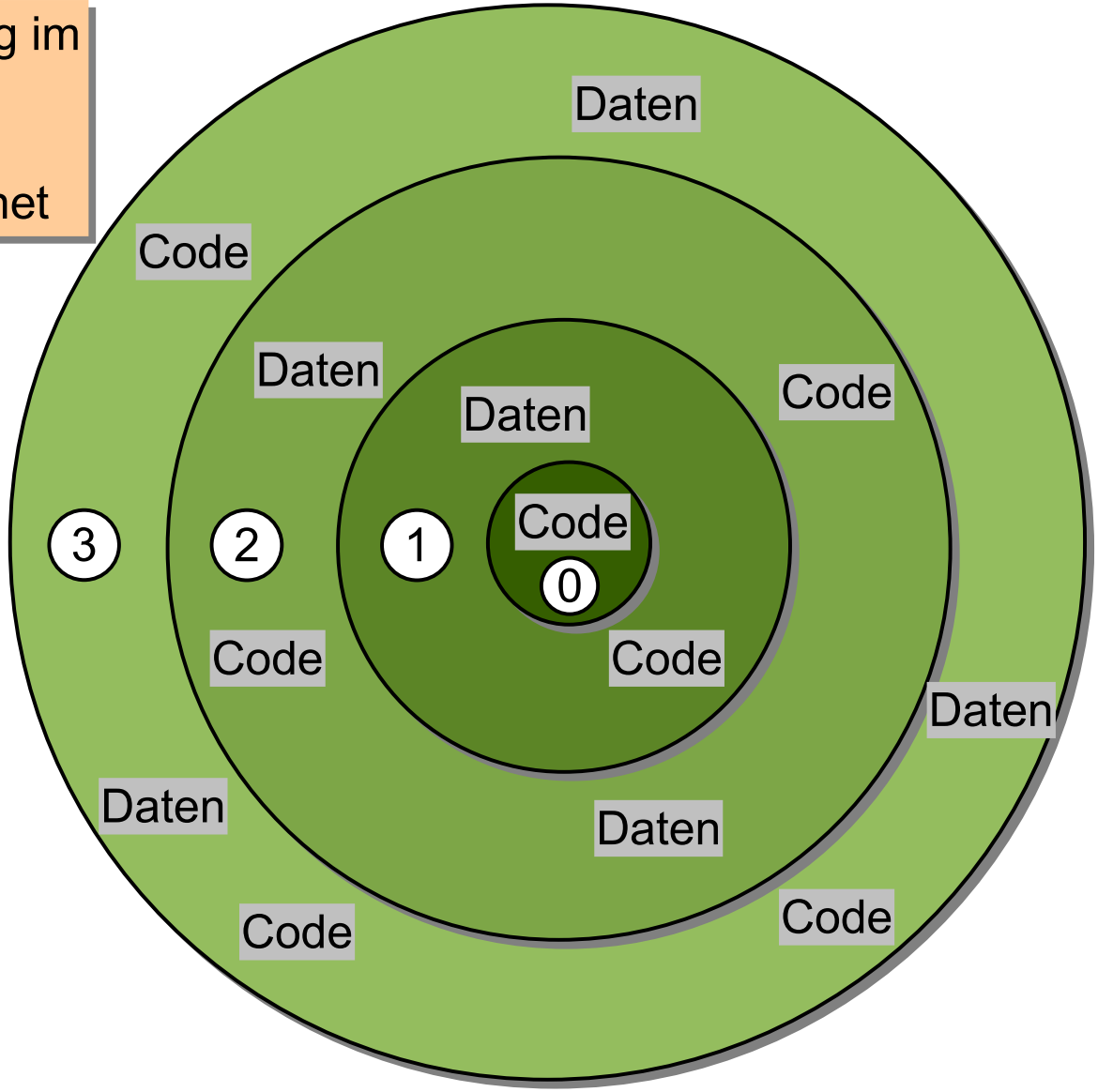
- Problem: bei aktiver *Paging Unit* wäre eine IA-32 CPU erheblich langsamer, wenn bei jedem Speicherzugriff das *Page Directory* und die *Page Table* gelesen werden müssten
- Lösung: der ***Translation Lookaside Buffer*** (TLB):
  - vollassoziativer Cache
    - Tag: 20 Bit Wert aus *Page Directory* und *Page Table* Index
    - Daten: *Page Frame* Adresse
    - Größe beim 80386: 32 Einträge
  - bei normalen Anwendungen erreicht der TLB eine Trefferrate von etwa 98%
  - Schreiben in das CR3 Register invalidiert den TLB

Schutz

- die wichtigste Eigenschaft des IA-32 Protected Mode ist das Schutzkonzept
- **Ziel:** fehlerhaften oder nicht vertrauenswürdigen Code isolieren
  - Schutz vor Systemabstürzen
  - Schutz vor unberechtigten Datenzugriffen
  - keine unberechtigten Operationen, z.B. I/O Port Zugriffe
- Voraussetzungen: Code und Daten ...
  - werden hinsichtlich der Vertrauenswürdigkeit kategorisiert
  - bekommen einen Besitzer (siehe "Multitasking")

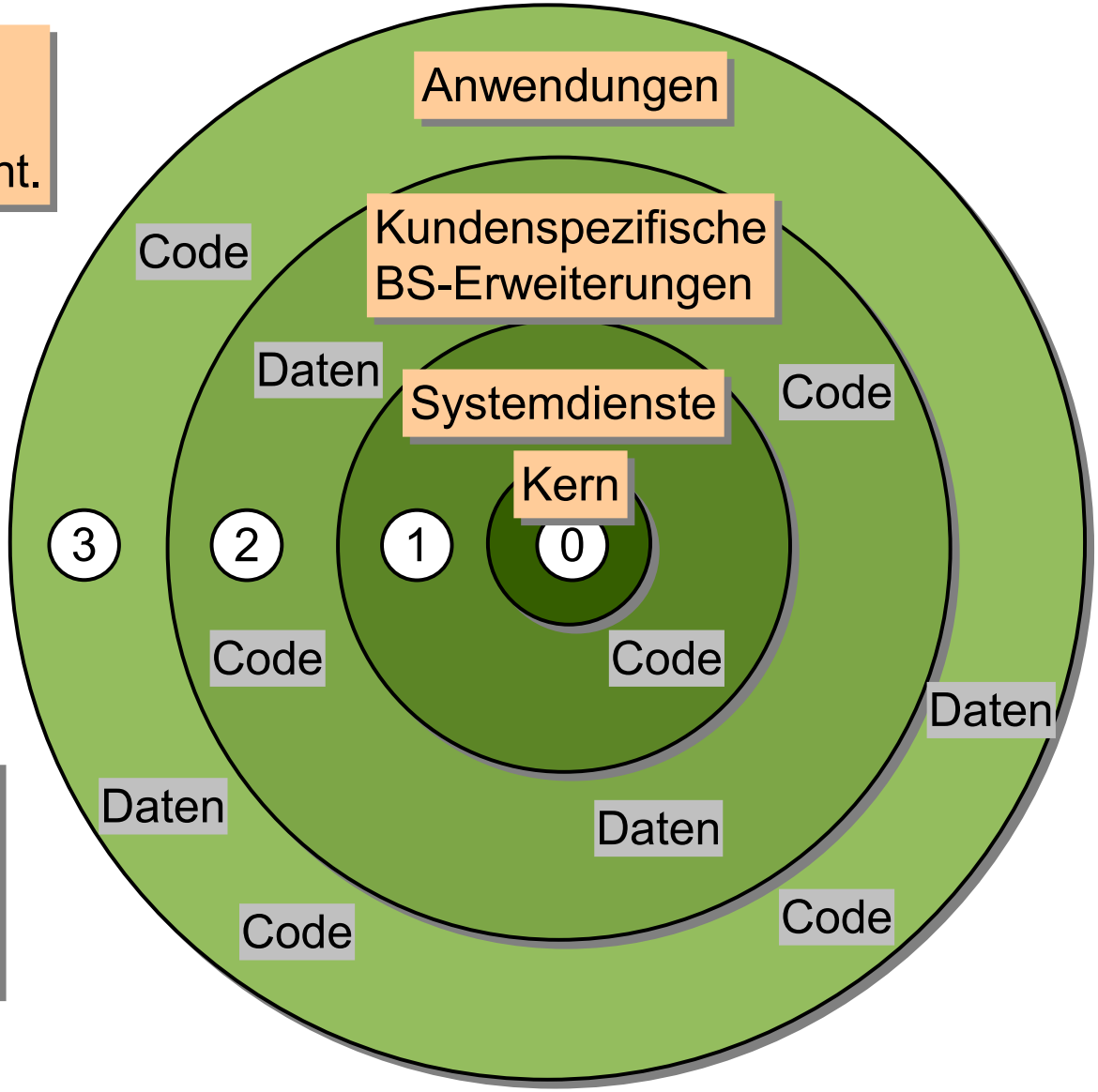
Schutzringe und *Gates*

Durch einen 2 Bit Eintrag im Segmentdeskriptor wird jedes Segment einer Privileg-Ebene zugeordnet



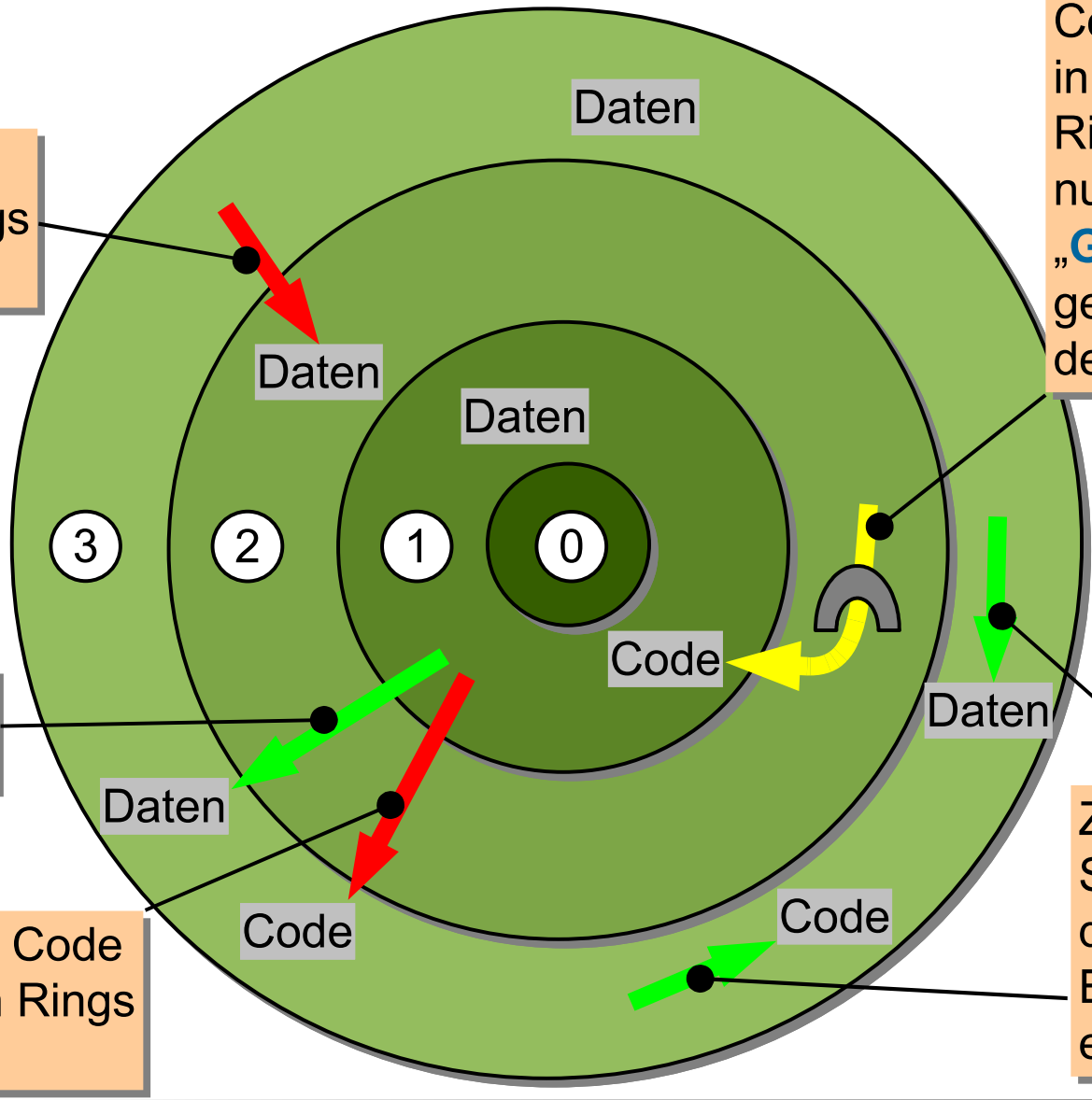
Schutzringe und *Gates*

Privileg-Ebene 3 ist die niedrigste und für Anwendungen gedacht.



Privileg-Ebene 0 ist die höchste und dem Betriebssystem*kern* vorbehalten.

Schutzringe und *Gates*



Zugriff auf Daten eines inneren Rings ist verboten

Code eines inneren Rings kann nur durch „**Gates**“ aufgerufen werden

Zugriff auf Daten eines äußeren Rings erlaubt

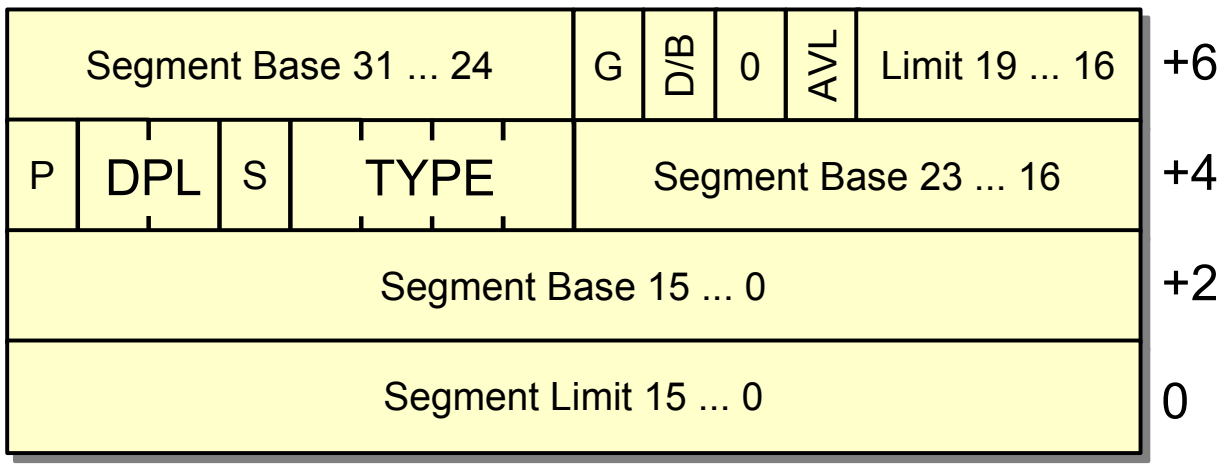
der Aufruf von Code eines äußeren Rings ist verboten!

Zugriffe auf Segmente der selben Ebene sind erlaubt

Segmentdeskriptoren

- weitere Informationen über die Schutzanforderungen der Segmente enthalten die Deskriptoren
  - jede Verletzung führt zum Auslösen einer Ausnahme

ein **Segment-Deskriptor**



TYPE – Data:

ED	- Expansion Direction
W	- Writable
A	- Accessed

TYPE – Code:

C	- Conforming
R	- Readable
A	- Accessed

P	- Present Bit
DPL	- Descriptor Privilege Level
S	- System Segment

G	- Granularity
D/B	- 16/32 Bit Seg.
AVL	- Available for OS

Beispiel: Das "flache" Speichermodell

- die meisten PC Betriebssysteme nutzen die Segmentierung nicht.
  - 32 Bit *Offset* der logischen Adresse = lineare Adresse
  - trotzdem müssen zwei Segmentdeskriptoren angelegt werden:

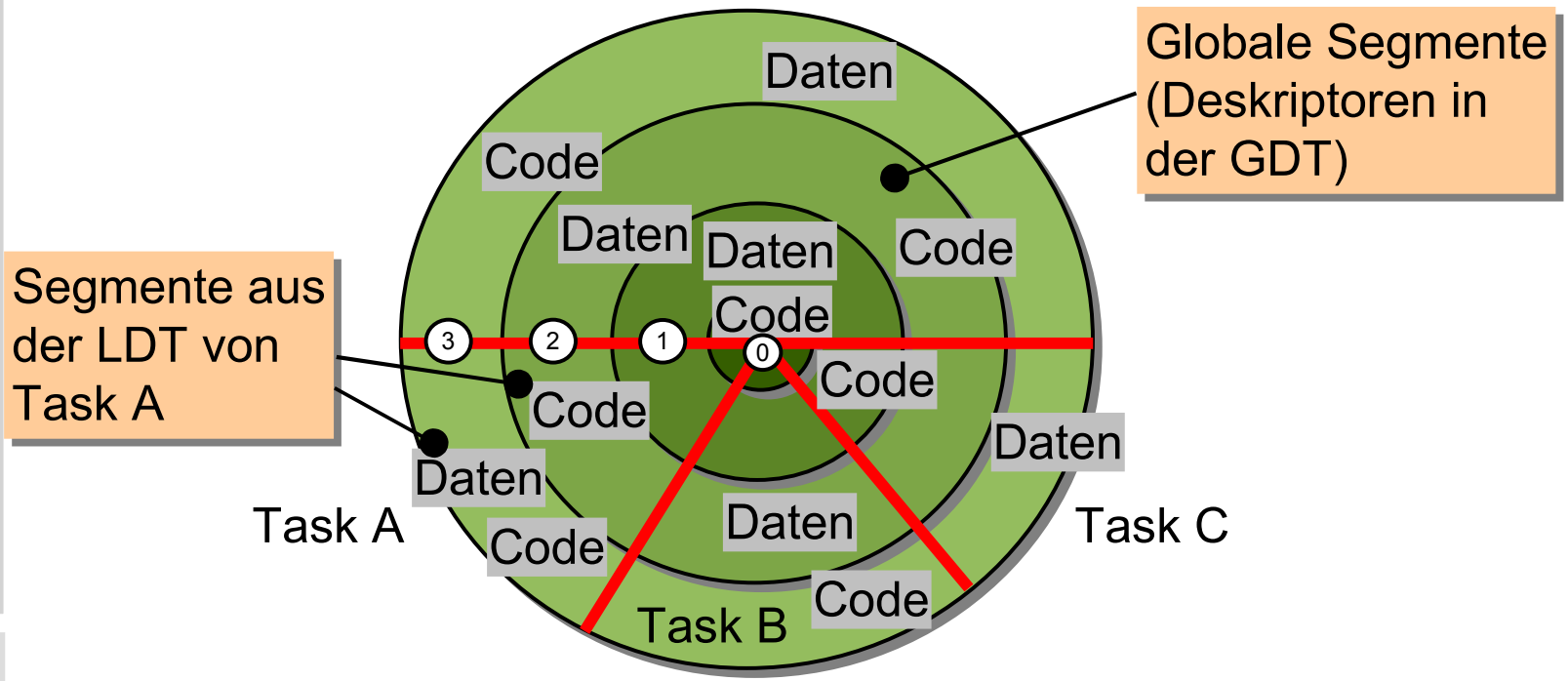
```
;
; Descriptor-Tabellen
;
gdt:
        dw      0,0,0,0   ; NULL Deskriptor

        dw      0xFFFF    ; 4Gb - (0x100000*0x1000 = 4Gb)
        dw      0x0000    ; base address=0
        dw      0x9A00    ; code read/exec
        dw      0x00CF    ; granularity=4096,
                          ; 386 (+5th nibble of limit)

        dw      0xFFFF    ; 4Gb - (0x100000*0x1000 = 4Gb)
        dw      0x0000    ; base address=0
        dw      0x9200    ; data read/write
        dw      0x00CF    ; granularity=4096,
                          ; 386 (+5th nibble of limit)
```

Agenda

Einordnung
Urvater: Der 8086
Die 32-Bit Intel-Architektur
Protected Mode
Multitasking
Zusammenfassung

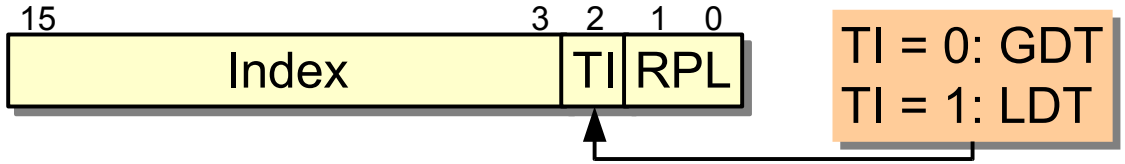
Multitasking

- neben dem Schutz vor unberechtigten "vertikalen" Zugriffen zwischen Segmenten unterschiedlicher Ebenen unterstützt IA-32 auch ein Task-Konzept ("horizontale Trennung“)
- die Zuordnung von Segmenten zu Tasks erfolgt über "Lokale Deskriptortabellen" (LDTs)



Lokale Segmentdeskriptortabellen

- ... sind nötig, wenn der Segmentselektor (z.B. aus einem Segmentregister) sich auf die LDT bezieht:



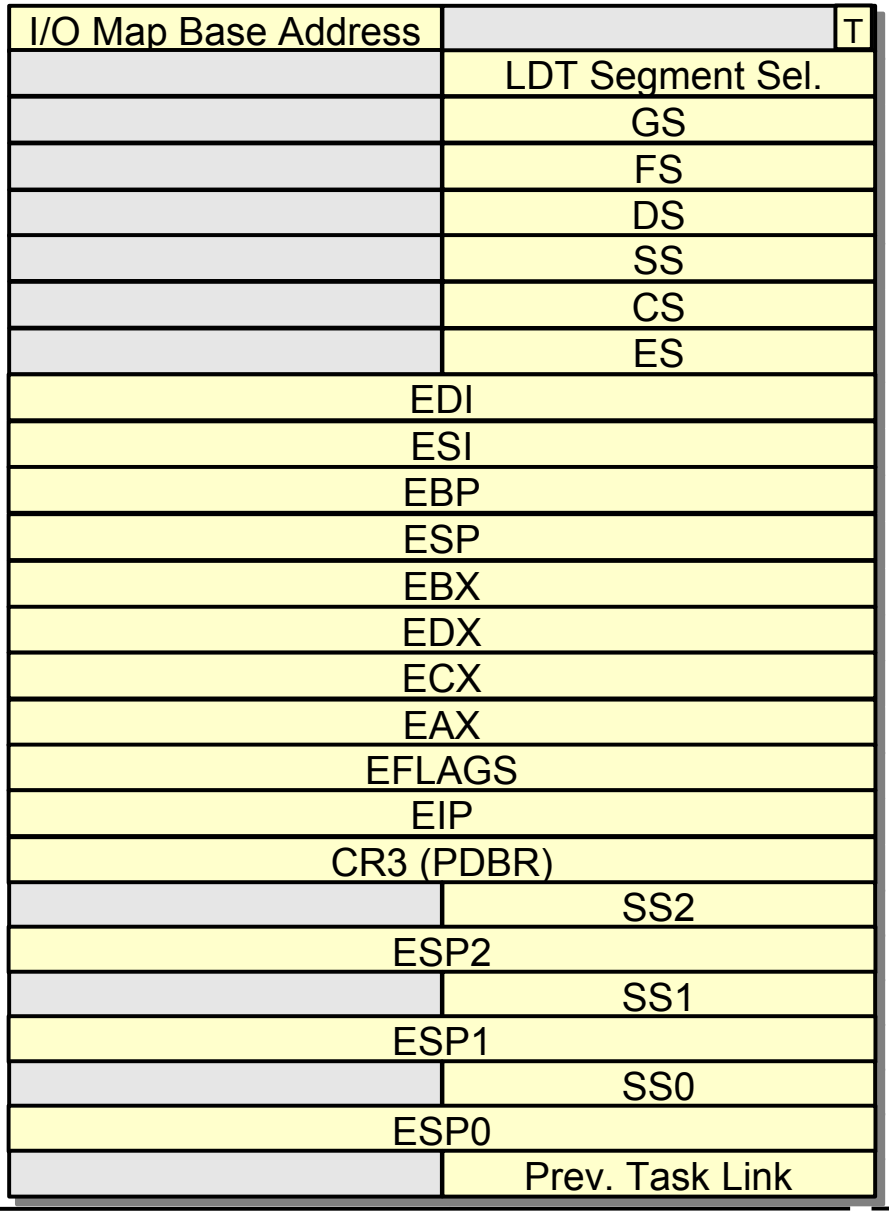
- ... werden mit Hilfe des **LDTR** gefunden, das bei jedem Taskwechsel ausgetauscht wird:

Speicherverwaltungsregister

	15 … 0	31 … 0	19 … 0
TR	TSS-Sel.	TSS-Basisadresse	TSS-Limit
LDTR	LDT-Sel.	LDT-Basisadresse	LDT-Limit
IDTR		IDT-Basisadresse	IDT-Limit
GDTR		GDT-Basisadresse	GDT-Limit

Der *Task*-Zustand: TSS Segmente

- das Task-Register TR verweist auf eine Datenstruktur, die den kompletten Task-Zustand aufnimmt
- bei einem Task-Wechsel (siehe nächste Seite) wird der komplette Zustand gesichert und der Zustand des Ziel-Tasks geladen
  - alles in Hardware!



Task-Wechsel

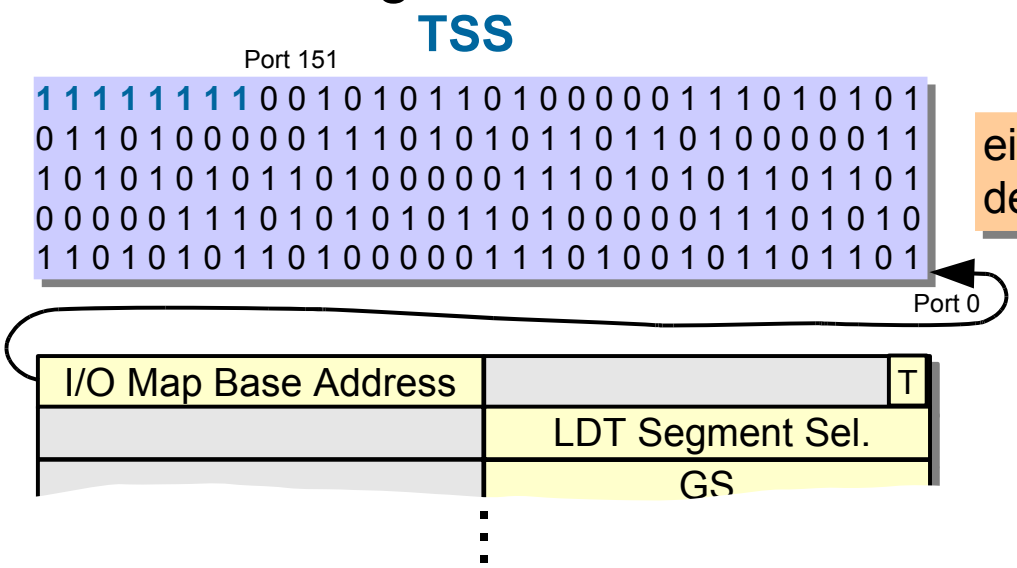
- für einen Task-Wechsel benötigt man entweder ...
  - ein Task-Gate in der GDT, einer LDT oder der IDT (Task-Wechsel bei Unterbrechungen!)
  - oder einfach nur einen TSS Deskriptor in der GDT
- ausgelöst werden kann ein Wechsel durch ...
  - eine JMP Instruktion
  - eine CALL Instruktion
  - eine Unterbrechung
  - eine IRET Instruktion
- Nested Tasks
  - bei Unterbrechungen und CALLs wird das NT Flag im EFLAGS Register und der "Prev. Task Link" im TSS gesetzt.
  - Wenn dies der Fall ist, springt IRET zum vorherigen Task zurück.

Ein-/Ausgaben im Protected Mode

- nicht jeder beliebige Task darf Ein-/Ausgabe durchführen!
- Zugriffe auf Geräte im Speicher (memory-mapped I/O) können über Speicherschutz abgefangen werden
- Zugriffe auf I/O Ports werden eingeschränkt:
  - die I/O Privilege Level Bits im EFLAGS Register erlauben Ein- und Ausgaben auf bestimmten Schutzringen
  - auf den anderen Ebenen regelt die I/O Permission Bitmap für jeden Task und Port der Zugriff:

Ports mit größeren Nummern dürfen nicht angesprochen werden

den Abschluss bildet immer ein Byte mit 0xff



eine '1' verhindert den Portzugriff

IA-32: Was gibt es sonst noch?

- Physical Address Extension (PAE)
  - ab Pentium Pro: 36-Bit Adressen (physikalisch)
  - erweiterte *Page Table* Einträge
  - weitere *Page Directory* Ebene
- System Management Mode (SMM)
  - gibt dem BIOS Kontrolle über das System
  - das Betriebssystem merkt davon nichts!
- Virtualisierung der CPU
  - der Virtual 8086 Mode
    - 16 Bit Anwendungen oder Betriebssysteme laufen als IA-32 Task in einer geschützten Umgebung
  - Vanderpool Technology
    - Hardwareunterstützung für virtuelle Maschinenlösungen wie VmWare, VirtualPC oder Xen
    - erlaubt die Ausführung von E0 Protected Mode Code in einer VM

Agenda

Einordnung
Urvater: Der 8086
Die 32-Bit Intel-Architektur
Protected Mode
Multitasking
Zusammenfassung

Zusammenfassung

- die IA-32 Architektur ist ausgesprochen komplex
  - segmentbasierter und seitenbasierter Speicherschutz
  - Hardwareunterstützung für Multitasking
  - Task-Aktivierung bei Unterbrechungen
  - Schutz von I/O Ports pro Task
  - ...
- viele dieser Features werden von heutigen Betriebssystemen nicht genutzt
  - typisch ist der flache Adressraum mit aktiver Paging Unit
  - Hardware-Tasks sind kaum portierbar
- bemerkenswert ist auch die konsequente Kompatibilität mit älteren Prozessorversionen
  - Stichwort "PIC" und "A20 Gate"! 🙂

Zusammenfassung

- die
  - se
  - H
  - Ta
  - S
  - ...
- viel
  Bet
  - ty
  - H
- ber
  mit
  - Stichwort "PIC" und "A20 Gate"!

Was ist mit 64-Bit? ⇝ AMD64 (x86-64)

- 64-Bit-Erweiterung von x86, propagiert von AMD
  - offizielle Intel-Bezeichnung: x86-64
  - **Achtung:** IA-64 bezeichnet Intels (tote?) Itanium-Architektur
- Wesentliche Features
  - 64-Bit-Register und -Adressen
  - 16 (statt bisher 8) General-Purpose Register
- Viele IA-32-Features sind weggefallen!
  - Segmentierung (außer FS und GS Register)
    ⇝ MMU *muss* benutzt werden
  - Task-Modell
  - Ring-Modell (außer -1, 0, 3)
  - Virtual-X86-Modus

Betriebssysteme (BS)

VL 8 – Koroutinen und Fäden

Volkmar Sieh / **Daniel Lohmann**

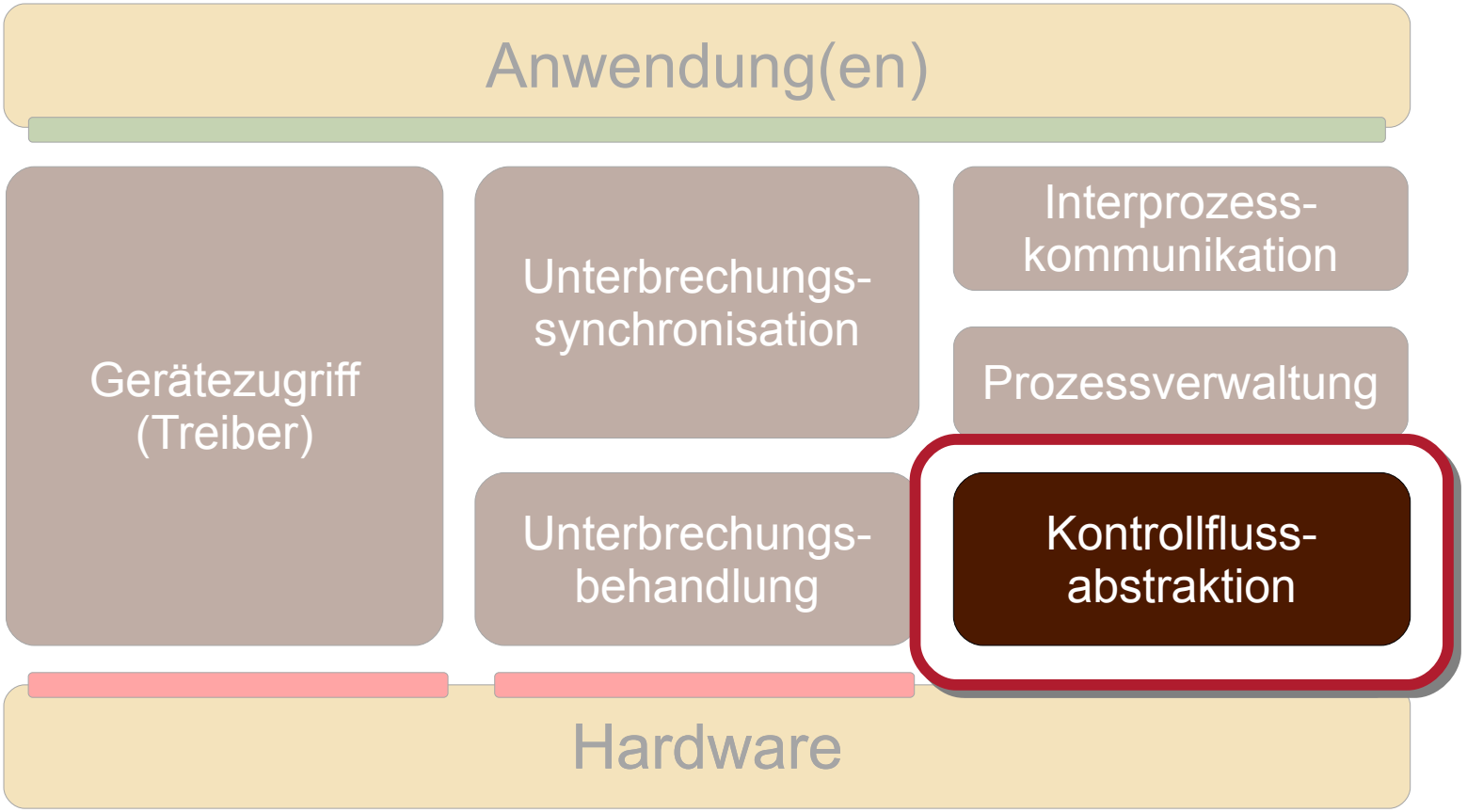
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 13. Dezember 2023

`https://sys.cs.fau.de/lehre/ws23/bs`

Überblick: Einordnung dieser VL



Agenda

Motivation
Grundbegriffe
Implementierung
Ausblick
Zusammenfassung
Referenzen

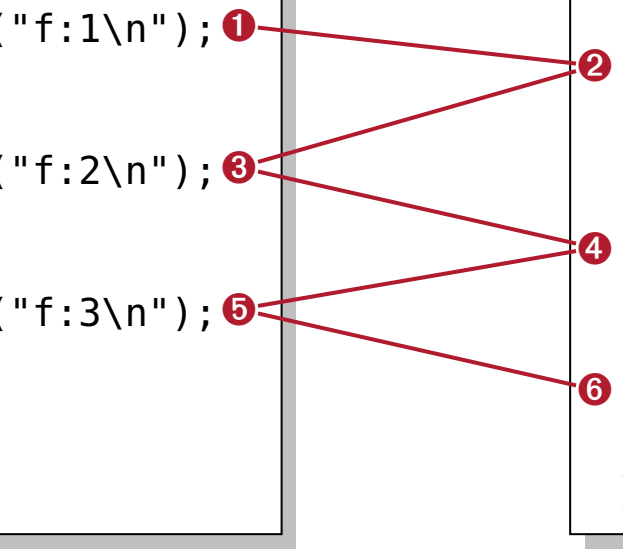
Agenda

- Motivation
  - Einige Versuche
  - Fazit
- Grundbegriffe
- Implementierung
- Ausblick
- Zusammenfassung
- Referenzen

Motivation: Quasi-Parallelität

```
void f() {
    printf("f:1\n");
    printf("f:2\n");
    printf("f:3\n");
}
```

```
void g() {
    printf("g:A\n");
    printf("g:B\n");
    printf("g:C\n");
}
```




```
int main() {
    ?
}
```

- **Gegeben:** Funktionen `f()` und `g()`
- **Ziel:** `f()` und `g()` sollen „verschränkt” ablaufen

Im Folgenden einige Versuche...

Motivation: Quasi-Parallelität


```
void f() {
    printf("f:1\n");

    printf("f:2\n");

    printf("f:3\n");


}
```

```
void g() {

    printf("g:A\n");


    printf("g:B\n");

    printf("g:C\n");

}
```

```
int main() {


  f();
  g();
}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
f:2
f:3
g:A
g:B
g:C
```

So funktioniert es natürlich nicht.

Motivation: Quasi-Parallelität


```
void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();


}
```

```
void g() {

    printf("g:A\n");



    printf("g:B\n");


    printf("g:C\n");

}
```

```
int main() {


  f();

}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
g:A
g:B
g:C
f:2
...
```

So geht es wohl auch nicht.

Motivation: Quasi-Parallelität


```
void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();


}
```

```
void g() {

    printf("g:A\n");
    f();


    printf("g:B\n");
    f();

    printf("g:C\n");
    f();
}
```

```
int main() {


  f();

}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
g:A
f:1
g:A
...
Segmentation fault
```

So **schon gar nicht!**

Motivation: Quasi-Parallelität


```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;


}
```

```
void g_start() {

    printf("g:A\n");
    g = &&l1; goto *f;


l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
void (*volatile f)();
void (*volatile g)();
int main() {
  f=f_start;
  g=g_start;
  f();

}
```

Und so?

Motivation: Quasi-Parallelität


```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;


}
```

```
void g_start() {

    printf("g:A\n");
    g = &&l1; goto *f;


l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
void (*volatile f)();
void (*volatile g)();
int main() {
  f=f_start;
  g=g_start;
  f();

}
```

```
lohmann@faui48a>gcc-2.95 -fomit-frame-
pointer -o coroutine coroutine.c
lohmann@faui48a>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Klappt!

Motivation: Quasi-Parallelität


```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;


}
```

```
void g_start() {

    printf("g:A\n");
    g = &&l1; goto *f;


l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
void (*volatile f)();
void (*volatile g)();
int main() {
  f=f_start;
  g=g_start;
  f();
}
```

```
lohmann@faui48a>gcc-2.95 -fomit-frame-
pointer -o coroutine coroutine.c
lohmann@faui48a>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Bitte nicht zu Hause nachmachen!

Quasi-Parallelität: Feststellungen

- C/C++ bietet keine Bordmittel für „verschränkte” Ausführung
  - einfache Funktionsaufrufe (Versuche 1 und 2)
    - laufen immer komplett durch (*run-to-completion*)
  - rekursive Funktionsaufrufe (Versuch 3)
    - dito, $\leadsto$ Endlosrekursion und Stapelüberlauf
- Wir brauchen Systemunterstützung, um Kontrollflüsse „während der Ausführung” verlassen und wieder betreten zu können
  - ungefähr so wie in Versuch 4
    - „Fortsetzungs”-PC wird gespeichert, mit goto wieder aufgenommen
  - aber bitte ohne die damit einhergehenden Probleme!
    - *computed gotos* aus Funktionen sind undefiniert
    - Zustand besteht aus mehr als dem PC – was ist mit Registern, Stapel, . . .

Anmerkung: Aus Systemsicht („von unten”) würde der PC reichen!

- (PC) $\Leftrightarrow$ *minimaler Kontrollflusszustand*
- alles weitere ist letztlich eine Entwurfsentscheidung des **Compilers** $\leadsto$ [UE1]
- wird in der Praxis jedoch durch Hardwarehersteller nahegelegt (ISA, ABI)

Agenda

Motivation

Grundbegriffe

- Routine und asymmetrisches Fortsetzungsmodell
- Koroutine und symmetrisches Fortsetzungsmodell

Implementierung

Ausblick

Zusammenfassung

Referenzen

Grundbegriffe: Routine, Kontrollfluss

- Routine: eine endliche Sequenz von Anweisungen
  - z. B. die Funktion f
  - Sprachmittel fast aller Programmiersprachen
  - wird ausgeführt durch (Routinen-)Kontrollfluss
- (Routinen-)Kontrollfluss: eine Routine in Ausführung
  - Ausführung und Kontrollfluss sind synonme Begriffe
  - z. B. die Ausführung <f> der Funktion f
    - beginnt bei Aktivierung mit der ersten Anweisung von f

Zwischen Routinen und Ausführungen besteht eine Schema–Instanz Relation. Zur klaren Unterscheidung werden die Instanzen ($\mapsto$ Ausführungen) deshalb hier in spitzen Klammern gesetzt:

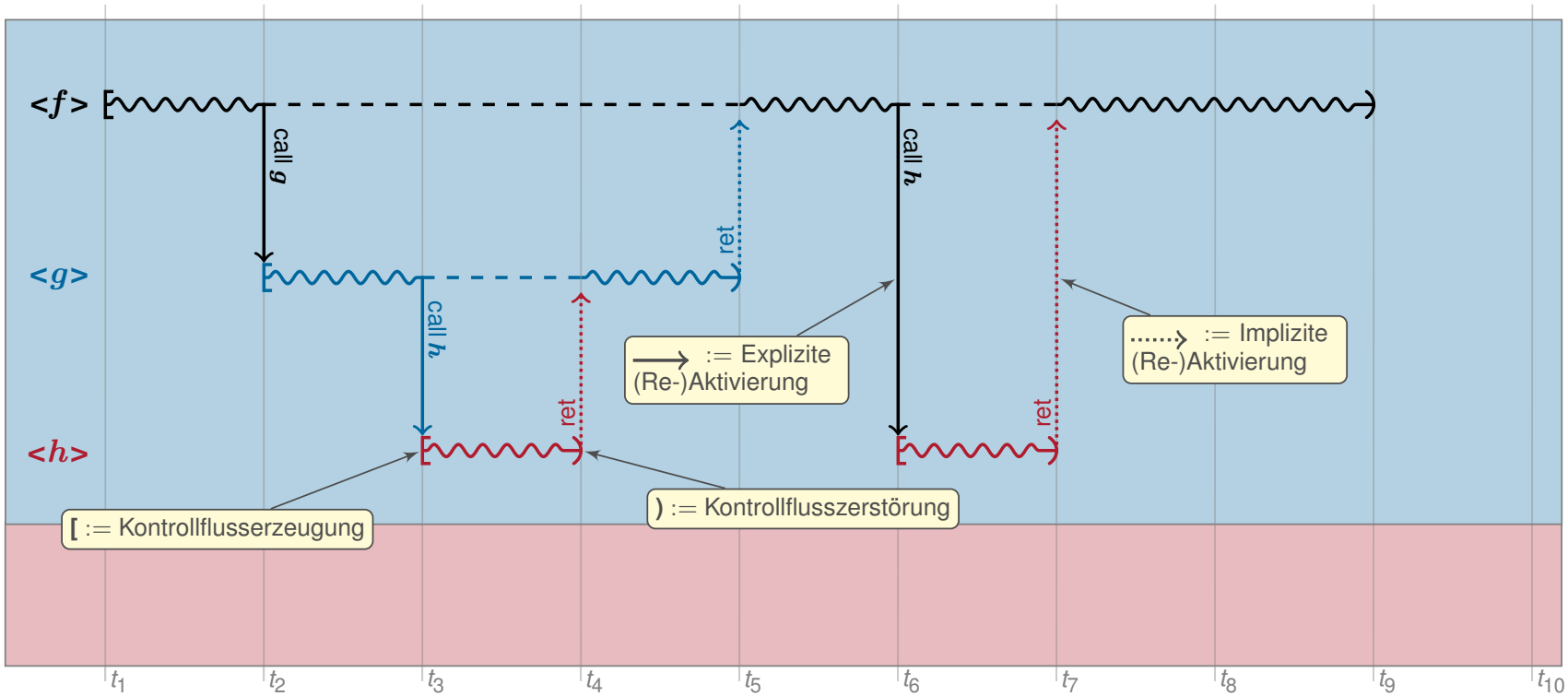
<f>, <f'>, <f''> sind Ausführungen von f.

Grundbegriffe: Routine, Kontrollfluss

- Routinen-Kontrollflüsse werden erzeugt, gesteuert, und zerstört mit speziellen Elementaroperationen
  - <f> *call* g (Ausführung <f> erreicht Anweisung `call g`)
    1. **erzeugt** neue Ausführung <g> von g
    2. **suspendiert** die Ausführung <f>
    3. **aktiviert** die Ausführung <g>
       ($\rightsquigarrow$ erste Anweisung wird ausgeführt)
  - <g> *ret* (Ausführung <g> erreicht Anweisung `ret`)
    1. **suspendiert** die Ausführung <g>
    2. **zerstört** die Ausführung <g>
    3. **reaktiviert** die Ausführung des Vater-Kontrollflusses (z. B. <f>)

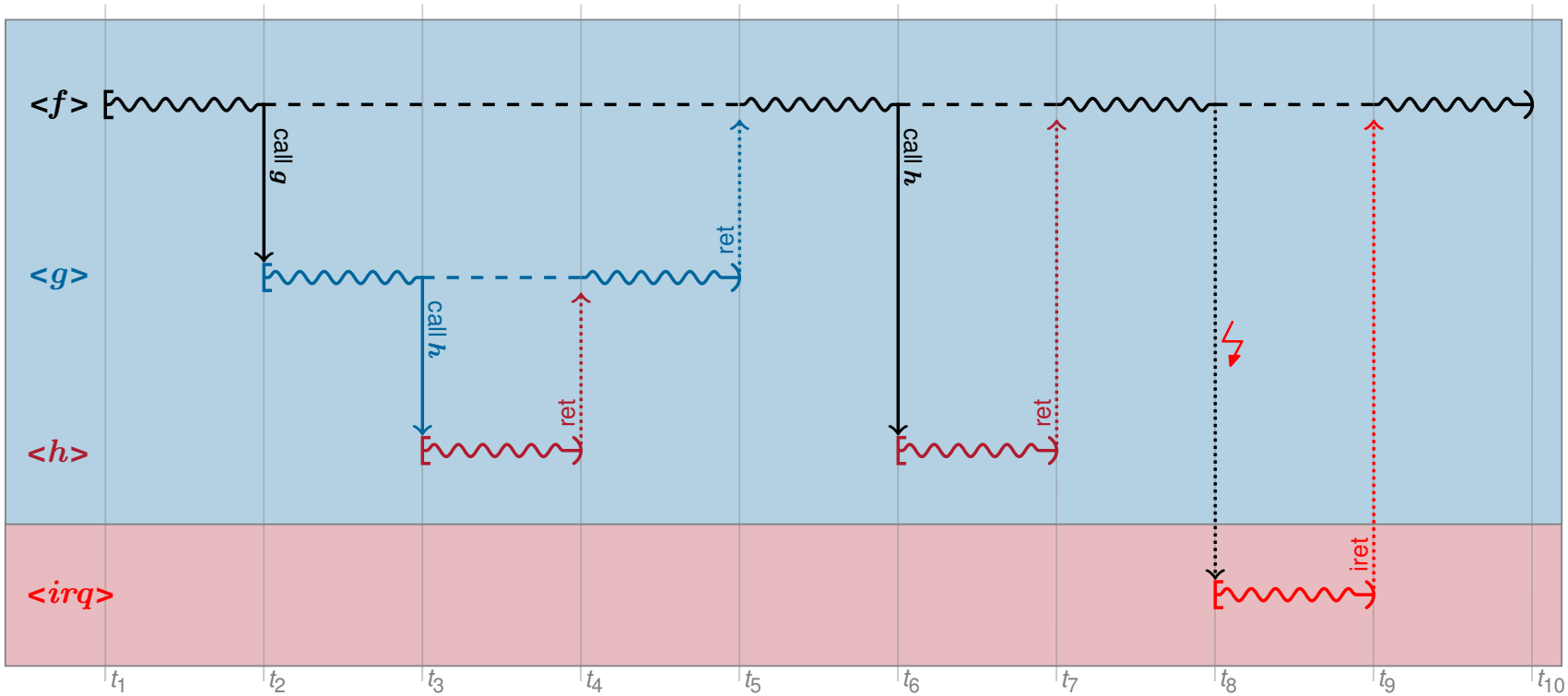
Routinen ↦ asymmetrisches Fortsetzungsmodell

- Routinen-Kontrollflüsse bilden eine Fortsetzungshierarchie
  - Vater–Kind Relation zwischen Erzeuger und Erzeugtem
- Aktivierte Kontrollflüsse werden nach LIFO fortgesetzt
  - Der zuletzt aktivierte Kontrollfluss terminiert immer zuerst
  - Vater wird erst fortgesetzt, wenn Kind terminiert



Routinen ↦ asymmetrisches Fortsetzungsmodell

- Das gilt auch bei Unterbrechungen
  - $<f>$ ↯ irq ist wie *call*, nur implizit
- Unterbrechungen können als implizit erzeugte und aktivierte Routinen-Ausführungen verstanden werden



Grundbegriffe: Koroutine

- Koroutine (engl. *Coroutine*): verallgemeinerte Routine [1]
  - erlaubt zusätzlich: expliziten Austritt und Wiedereintritt
  - Sprachmittel *einiger* Programmiersprachen
    - z. B. Modula-2, Simula-67, Stackless Python
  - wird ausgeführt durch Koroutinen-Kontrollfluss
- Koroutinen-Kontrollfluss: eine Koroutine in Ausführung
  - Kontrollfluss mit eigenem, unabhängigen Zustand
    - mindestens Programmzähler (PC)
    - zusätzlich je nach (zu unterstützendem) Compiler / ABI / ISA: weitere Register, Stapel, ...
    - Im Prinzip ein eigenständiger Faden (engl. *Thread*) – **dazu später mehr**

Koroutinen und Koroutinen-Kontrollflüsse stehen ebenfalls in einer Schema–Instanz Relation.

In der Literatur ist diese Unterscheidung unüblich $\rightsquigarrow$ Koroutinen-Kontrollflüsse werden (vereinfacht) ebenfalls als Koroutinen bezeichnet.

Grundbegriffe: Koroutine

- Koroutinen-Kontrollflüsse werden erzeugt, gesteuert, und zerstört über zusätzliche Elementaroperationen
  - *create* g
    1. **erzeugt** neue Korotinen-Ausführung <g> von g
  - <f> *resume* <g>
    1. **supendiert** die Koroutinen-Ausführung <f>
    2. **(re-)aktiviert** die Koroutinen-Ausführung <g>
  - *destroy* <g>
    1. **zerstört** die Koroutinen-Ausführung <g>

Unterschied zu Routinen-Kontrollflüssen: [SP, C 10-8]

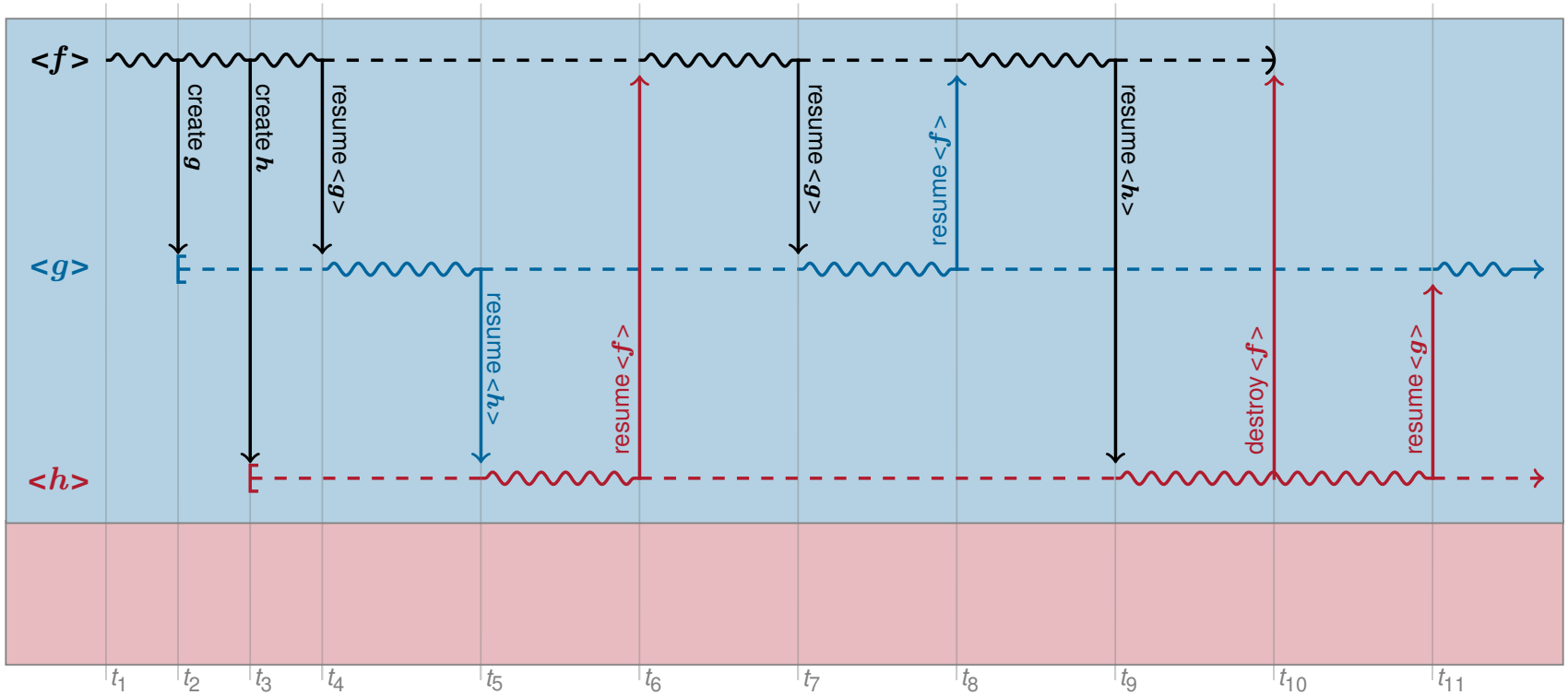
Aktivierung und Reaktivierung sind

zeitlich entkoppelt von Erzeugung und Zerstörung.

$\rightsquigarrow$ Koroutinen sind **echt mächtiger** als Routinen.

Koroutinen ↦ symmetrisches Fortsetzungsmodell

- Koroutinen-Kontrollflüsse bilden eine Fortsetzungsfolge
  - Koroutinenzustand bleibt über Ein-/Austritte hingweg erhalten
- Alle Koroutinen-Kontrollflüsse sind gleichberechtigt
  - kooperatives Multitasking
  - Fortsetzungsreihenfolge ist beliebig



Koroutinen und Programmfäden

- Koroutinen-Kontrollflüsse werden oft auch bezeichnet als
  - kooperative Fäden (engl. *cooperative Threads*)
  - Fasern (engl. Fibers)
- Das ist im Prinzip richtig, die Begriffe entstammen jedoch aus verschiedenen Welten
  - Koroutinen-Unterstützung ist historisch (eher) ein **Sprach**merkmal
  - Mehrfädigkeit ist historisch (eher) ein **Betriebssystem**merkmal
  - Die Grenzen sind fließend
    - *Sprachfunktion — (Laufzeit-)Bibliothekfunktion — Betriebssystemfunktion*
- Wir verstehen Koroutinen als technisches Konzept
  - um Mehrfädigkeit im BS zu implementieren
  - insbesondere später auch nicht-kooperative Fäden

Agenda

Motivation
Grundbegriffe
Implementierung
- Fortsetzungen
- Elementaroperationen

Ausblick
Zusammenfassung
Referenzen

Implementierung: Fortsetzungen

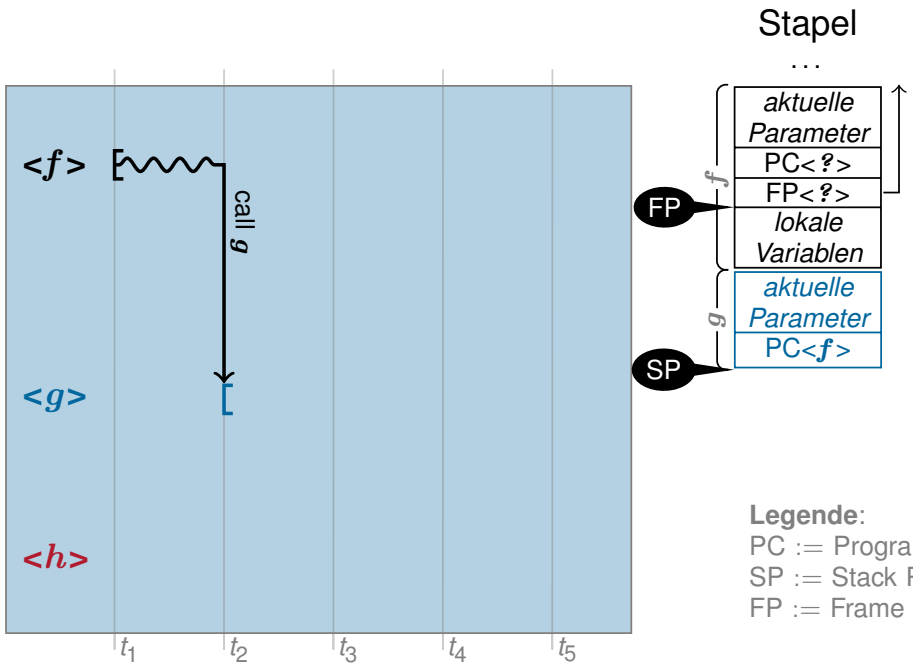
- Fortsetzung (engl. *Continuation*): Rest einer Ausführung
  - Eine Fortsetzung ist ein Objekt, das einen suspendierten Kontrollfluss repräsentiert.
    - Programmzähler, Register, lokale Variablen, ...
    - kurz: gesamter Kontrollflusszustand
  - wird benötigt, um den Kontrollfluss zu reaktivieren

Anmerkung: Fortsetzungen

- Continuations sind ursprünglich entstanden als ein Beschreibungsmittel der **denotationalen Semantik** [3].
- Sprachen wie Haskell oder Scheme bieten Continuations als eigenes Sprachmittel an.

Routinen ↦ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
  - in Form von Stapel-Rahmen, erzeugt und zerstört durch
    - Compiler (explizit) und CPU (implizit) bei *call*, *ret*
    - Kopplungsfunktion (explizit) und CPU (implizit) bei ϟ, *iret*
  - Der Compiler verwendet dafür i. a. den CPU-Stapel
    - `call`, `ret`, `push`, `pop`, ... verwenden implizit den CPU-Stapel

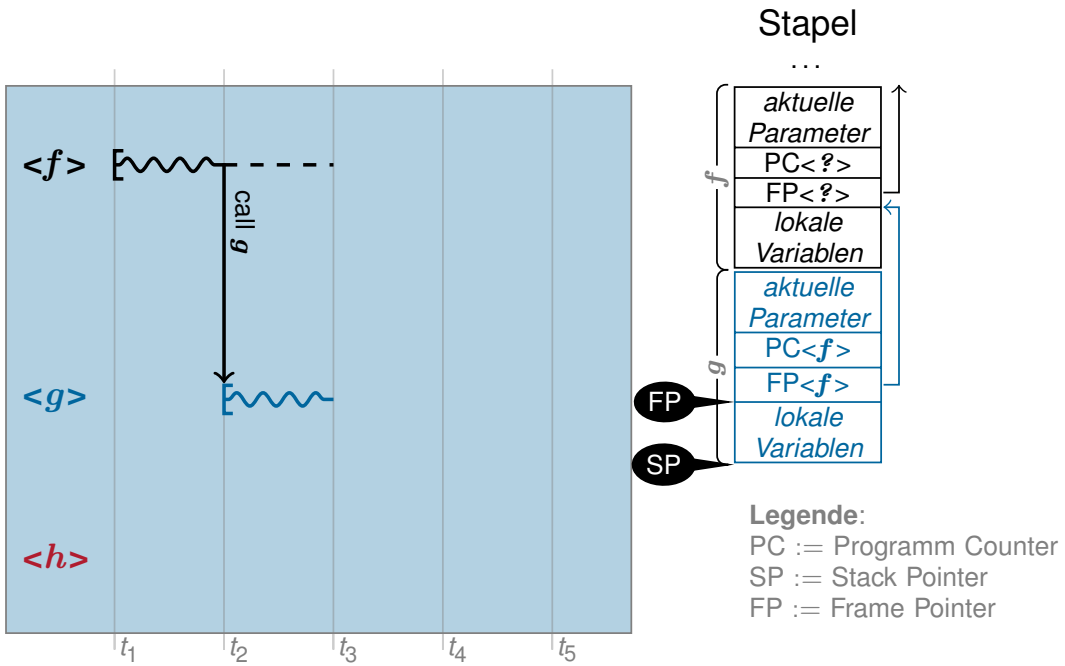


Legende:
PC := Programm Counter
SP := Stack Pointer
FP := Frame Pointer

Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.

Routinen ↦ asymmetrisches Fortsetzungsmodell

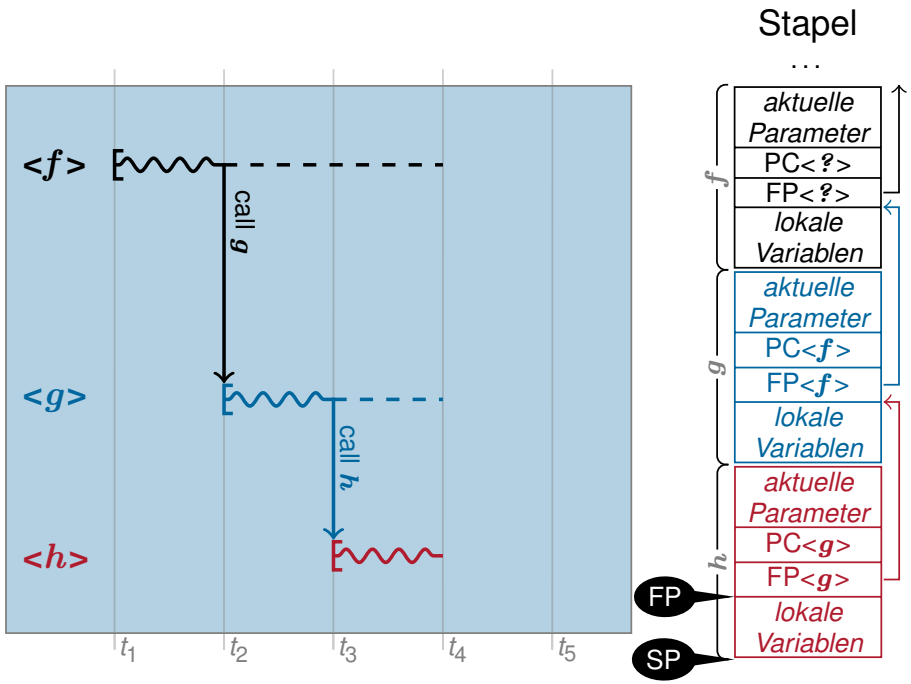
- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
  - in Form von Stapel-Rahmen, erzeugt und zerstört durch
    - Compiler (explizit) und CPU (implizit) bei *call*, *ret*
    - Kopplungsfunktion (explizit) und CPU (implizit) bei ϟ, *iret*
  - Der Compiler verwendet dafür i. a. den CPU-Stapel
    - `call`, `ret`, `push`, `pop`, ... verwenden implizit den CPU-Stapel



Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.

Routinen ↦ asymmetrisches Fortsetzungsmodell

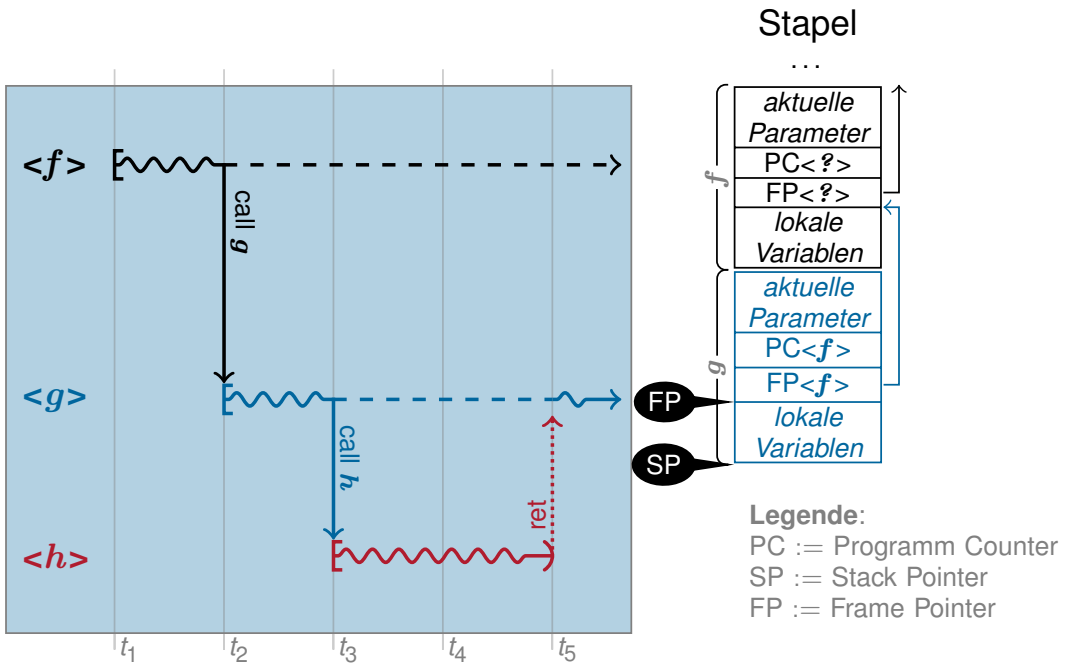
- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
  - in Form von Stapel-Rahmen, erzeugt und zerstört durch
    - Compiler (explizit) und CPU (implizit) bei *call*, *ret*
    - Kopplungsfunktion (explizit) und CPU (implizit) bei ϟ, *iret*
  - Der Compiler verwendet dafür i. a. den CPU-Stapel
    - `call`, `ret`, `push`, `pop`, ... verwenden implizit den CPU-Stapel



Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.

Routinen ↦ asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
  - in Form von Stapel-Rahmen, erzeugt und zerstört durch
    - Compiler (explizit) und CPU (implizit) bei *call*, *ret*
    - Kopplungsfunktion (explizit) und CPU (implizit) bei ϟ, *iret*
  - Der Compiler verwendet dafür i. a. den CPU-Stapel
    - `call`, `ret`, `push`, `pop`, ... verwenden implizit den CPU-Stapel



Für jeden Routinen-Kontrollfluss legen **CPU** und **Compiler** einen **Rahmen** an. Dieser enthält die **Fortsetzung** des Aufrufers.

Koroutinen $\mapsto$ symmetrisches Fortsetzungsmodell

- Koroutinen-Fortsetzungen werden i. a. nicht nativ unterstützt
- **Ansatz:** Koroutinen-Fortsetzungen durch Routinen-Fortsetzungen implementieren [2]
  - Ein *resume*-Aufruf sieht für den Compiler wie die Erzeugung und Aktivierung eines ganz normalen Routinen-Kontrollflusses aus.
  - Vor dem `ret` wird in *resume* jedoch intern der Koroutinen-Kontrollfluss gewechselt.
- **Folge:** Technisch gesehen, müssen wir das Routinen-Fortsetzungsmodell des Compilers bereitstellen
  - Registerverwendung $\leadsto$ nichtflüchte Register über Wechsel erhalten
  - Fortsetzungs-Stapel $\leadsto$ eigener Stapel für jede Koroutinen-Instanz

Eine Koroutinen-Instanz wird durch ihren Fortsetzungs-Stapel repräsentiert

- während der Ausführung ist dieser Stapel der CPU-Stapel
- oberster Stapel-Rahmen enthält immer die Fortsetzung
- Koroutinen-Wechsel $\mapsto$ Stapel-Wechsel + `ret`

Implementierung: *resume*

- Aufgabe: Koroutinen-Kontrollfluss wechseln

```
// Typ fuer Stapelzeiger (Stapel ist Feld von void*)
typedef void**  SP;

extern "C" void resume( SP& from_sp, SP& to_sp ) {
  /* aktueller Stapel-Rahmen ist Fortsetzung des zu
     suspendierenden Kontrollflusses (Aufrufer von resume) */

  < sichere CPU-Stapelzeiger in from_sp >
  < lade CPU-Stapelzeiger aus to_sp >

  /* aktueller Stapel-Rahmen ist Fortsetzung des zu
     reaktivierenden Kontrollflusses */

} // Ruecksprung
```

Problem: nicht-flüchtige Register

- Der Stapel-Rahmen enthält keine **nicht-flüchtigen Register**, da der Aufrufer davon ausgeht, dass diese nicht verändert werden.
- Wir springen jedoch in einen **anderen Aufrufer** zurück!

Implementierung: *resume*

- Problem: nicht-flüchtige Register
  - Routinen-Fortsetzung enthält keine nicht-flüchtigen Register
  - $\rightsquigarrow$ diese müssen explizit gesichert und restauriert werden
- Viele Implementierungsvarianten sind denkbar
  - nicht-flüchtige Register in eigener Struktur sichern ($\rightsquigarrow$ Übung)
  - oder einfach als „lokale Variablen” auf dem Stapel:

```
extern "C" void resume( SP& from_sp, SP& to_sp ) {
  /* aktueller Stapel-Rahmen ist Fortsetzung des zu
     suspendierenden Kontrollflusses (Aufrufer von resume) */
  <lege nicht-fluechtige Register auf den Stapel >
  < sichere CPU-Stapelzeiger in from_sp >
  < lade CPU-Stapelzeiger aus to_sp >
  <hole nicht-fluechtige Register vom Stapel >
  /* aktueller Stapel-Rahmen ist Fortsetzung des zu
     reaktivierenden Kontrollflusses */

} // Ruecksprung
```

Implementierung: *resume*

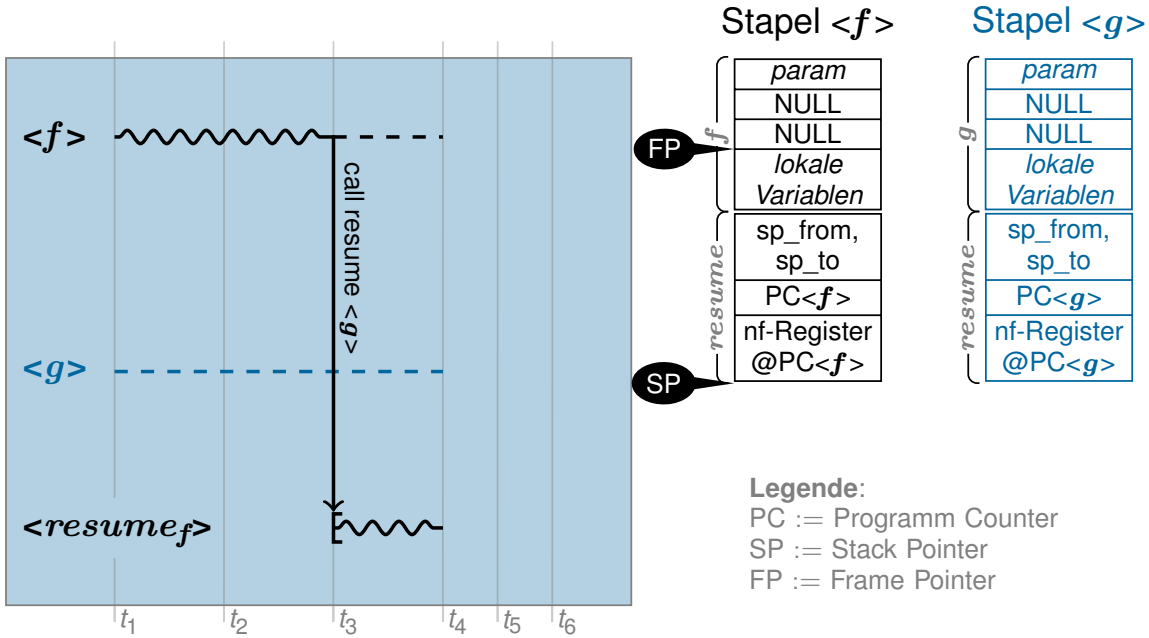
- Implementierung vom *resume* ist architekturabhängig
  - Aufbau der Stapel-Rahmen
  - nicht-flüchtige Register
  - Wachstumsrichtung des Stapels
- Außerdem muss man Register bearbeiten $\leadsto$ Assembler

Beispiel Motorola 68000:

```
// extern "C" void resume( SP& sp_from, SP& sp_to )
resume:
  move.l  4(sp), a0             // a0 = &sp_from
  move.l  8(sp), a1             // a1 = &sp_to
  movem.l d2-d7/a2-a6, -(sp)    // nf-Register auf den Stapel
  move.l  sp, (a0)              // sp_from = sp
  move.l  (a1), sp              // sp = sp_to
  movem.l (sp)+, d2-d7/a2-a6    // hole nf-Register vom Stapel
  rts                           // "Ruecksprung"
```

Beispiel: Verwendung von *resume*

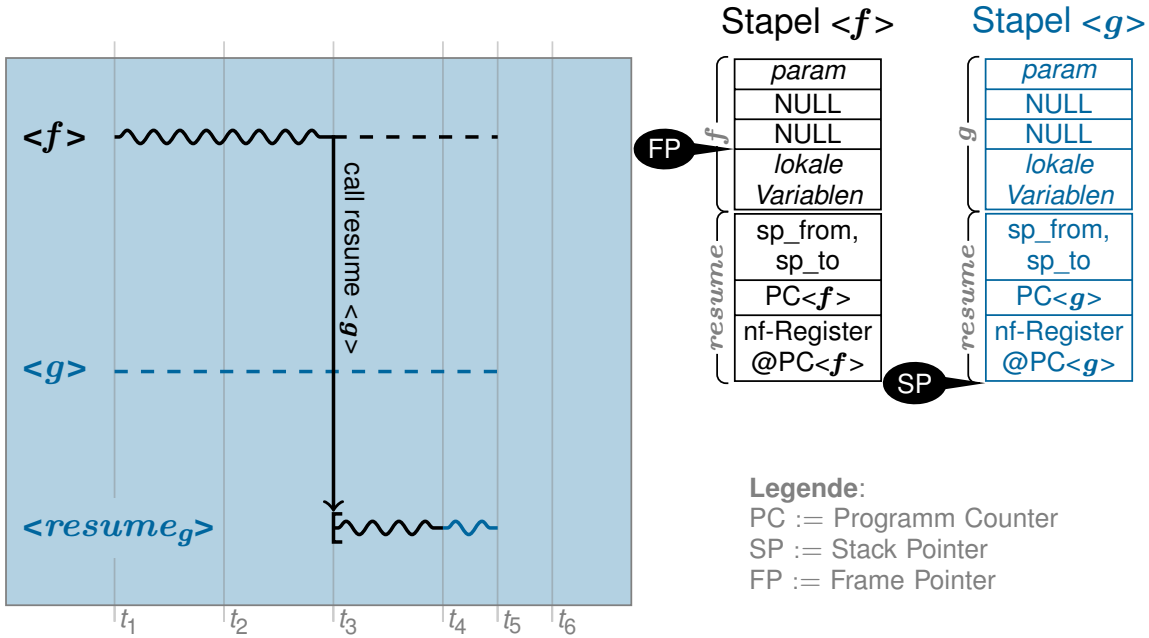
Koroutinen-Kontrollfluss <f> übergibt an <g>:



1. Koroutine <f> ist aktiv, Koroutine <g> ist suspendiert
2. <f> instantiiert den Routinen-Kontrollfluss <$resume_f$> und legt dazu Parameter (Stapelvariablen von <f> und <g>) sowie die Rücksprung-Adresse ($\mapsto$ Fortsetzung von <f>) auf den Stapel.
3. <$resume_f$> sichert nicht-flüchtige Register von <f> auf dem Stapel und eigenen SP in `sp_from`
4. Wechsel des SP auf den Stapel von <g> (`sp_to`) $\rightsquigarrow$ **Koroutinen-Wechsel**, nun läuft <$resume_g$>
5. <$resume_g$> holt nicht-flüchtige Register von <g> vom Stapel.
6. Routinen-Kontrollfluss <$resume_g$> terminiert mit `ret`: <g> ist aktiv, <f> ist suspendiert

Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss <f> übergibt an <g>:

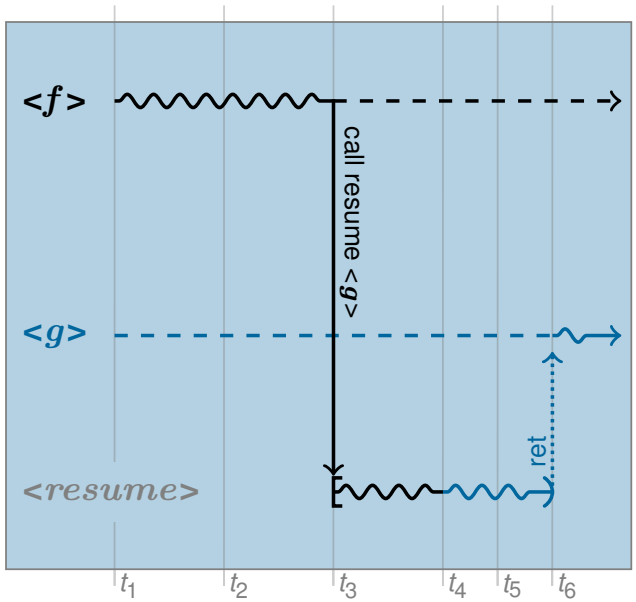


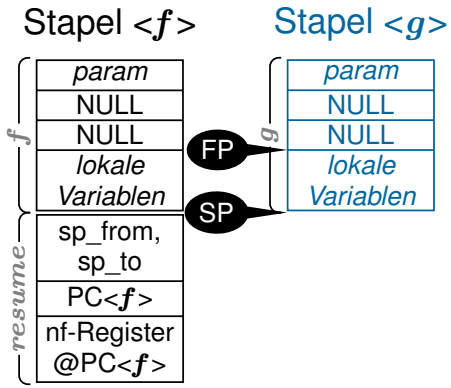
Legende:
PC := Programm Counter
SP := Stack Pointer
FP := Frame Pointer

1. Koroutine <f> ist aktiv, Koroutine <g> ist suspendiert
2. <f> instantiiert den Routinen-Kontrollfluss <$resume_f$> und legt dazu Parameter (Stapelvariablen von <f> und <g>) sowie die Rücksprung-Adresse ($\mapsto$ Fortsetzung von <f>) auf den Stapel.
3. <$resume_f$> sichert nicht-flüchtige Register von <f> auf dem Stapel und eigenen SP in `sp_from`
4. Wechsel des SP auf den Stapel von <g> (`sp_to`) $\rightsquigarrow$ **Koroutinen-Wechsel**, nun läuft <$resume_g$>
5. <$resume_g$> holt nicht-flüchtige Register von <g> vom Stapel.
6. Routinen-Kontrollfluss <$resume_g$> terminiert mit `ret`: <g> ist aktiv, <f> ist suspendiert

Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss <f> übergibt an <g>:





Legende:
PC := Programm Counter
SP := Stack Pointer
FP := Frame Pointer

1. Koroutine <f> ist aktiv, Koroutine <g> ist suspendiert
2. <f> instantiiert den Routinen-Kontrollfluss <$resume_f$> und legt dazu Parameter (Stapelvariablen von <f> und <g>) sowie die Rücksprung-Adresse ($\mapsto$ Fortsetzung von <f>) auf den Stapel.
3. <$resume_f$> sichert nicht-flüchtige Register von <f> auf dem Stapel und eigenen SP in `sp_from`
4. Wechsel des SP auf den Stapel von <g> (`sp_to`) $\rightsquigarrow$ **Koroutinen-Wechsel**, nun läuft <$resume_g$>
5. <$resume_g$> holt nicht-flüchtige Register von <g> vom Stapel.
6. Routinen-Kontrollfluss <$resume_g$> terminiert mit `ret`: <g> ist aktiv, <f> ist suspendiert

Implementierung: *create*

- Aufgabe: Koroutionen-Kontrollfluss <$start$> erzeugen
  - Gebraucht wird dafür
    1. **Stapelspeicher** (irgendwo, global)
    2. **Stapelzeiger**
    3. **Startfunktion**
    4. **Parameter** für die Startfunktion

```
static void* stack_start[ 256 ];
SP sp_start = &stack_start[ 256 ];
void start( void* param ) ...
```

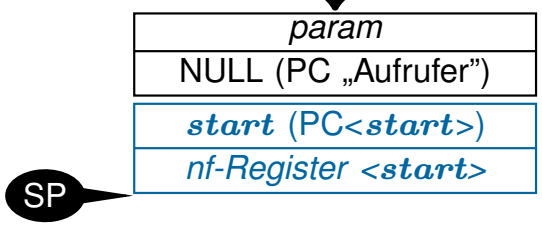
  - Koroutinen-Kontrollfluss wird suspendiert erzeugt
- Ansatz: *create* erzeugt zwei Stapel-Rahmen
  - so als hätte <$start$> bereits *resume* **als Routine** aufgerufen
    1. Rahmen der Startfunktion selber (erzeugt vom „virtuellen Aufrufer”)
    2. Rahmen von *resume* (enthält Fortsetzung in <$start$>)
  - erstes *resume* macht „Rücksprung” an den Beginn von $start$

Implementierung: *create*

Beispiel Motorola 68000:

```
void create( SP& sp_new, void (*start)(void*), void* param) {
  *(--sp_new) = param;  // Parameter von Startfunktion
  *(--sp_new) = 0;      // Aufrufer (gibt es nicht!)

  *(--sp_new) = start;  // Startadresse
  sp_new -= 11;         // nicht-fluechtige Register (Werte egal)
}
```

ergibt



Da der Rücksprung an den **Anfang** einer Funktion erfolgt, sind die Rahmen sehr einfach aufgebaut.

Zu diesem Fortsetzungspunkt hat ein Routinen-Kontrollfluss noch:

- keinen FP verwendet oder gesichert
- keine lokalen Variablen auf dem Stapel angelegt
- keine Annahmen über den Inhalt von nf-Registern

Implementierung: *destroy*

- Aufgabe: Koroutionen-Kontrollfluss zerstören
- Ansatz: Kontrollfluss-Kontext freigeben
  - entspricht Freigabe der Kontextvariablen ($\mapsto$ Stapelzeiger)
  - Stapelspeicher kann anschließend anderweitig verwendet werden

Das ist wenigstens mal einfach :-)

Agenda

Motivation
Grundbegriffe
Implementierung
Ausblick
- Koroutinen als Hilfsmittel für das BS
- Mehrfädrigkeit

Zusammenfassung
Referenzen

Ausblick: Betriebssystemfäden

- Koroutinen sind (eigentlich) ein Sprachkonzept
  - Multitasking auf Sprachebene
  - wir haben es hier für C/C++ (bzw. ein ABI) „nachgerüstet”
  - Kontextwechsel erfordert keine Systemprivilegien!

    $\rightsquigarrow$ muss also nicht zwingend im BS-Kern erfolgen
- Vorraussetzung für echtes Multitasking: Kooperation
  - Anwendungen müssen als Koroutinen implementiert sein
  - Anwendungen müssen sich gegenseitig kennen
  - Anwendungen müssen sich gegenseitig aktivieren
  - ...

Problem

Für uneingeschränkten Mehrprogramm-Betrieb ist das unrealistisch.

Ausblick: Betriebssystemfäden

Alternative: „Kooperationsfähigkeit” als Aufgabe des Betriebssystems auffassen

Ansatz: Anwendungen „unbemerkt” als eigenständige Fäden ausführen

- BS sorgt für die Erzeugung der Koroutinen-Kontrollflüsse
  - jede Anwendung wird als Routine aus einer BS-Koroutine aufgerufen
  - $\rightsquigarrow$ indirekt läuft jede Anwendung als Koroutine
- BS sorgt für die Suspendierung laufender Koroutinen-Kontrollflüsse
  - so dass Anwendungen nicht kooperieren müssen
  - erfordert einen Verdrängungsmechanismus
- BS sorgt für die Auswahl des nächsten Koroutinen-Kontrollflusses
  - so dass Anwendungen sich nicht gegenseitig kennen müssen
  - erfordert einen Scheduler

Mehr dazu in der nächsten Vorlesung!

Agenda

Motivation
Grundbegriffe
Implementierung
Ausblick
Zusammenfassung
Referenzen

Zusammenfassung: Quasi-Parallelität

- Ziel war die Ermöglichung von „Quasi-Parallelität”
  - Verschränkte Ausführung von Funktionen
    - Suspendierung und Reaktivierung von Funktions-Ausführungen
    - Begriff der Fortsetzung
- Routinen $\mapsto$ asymmetrisches Fortsetzungsmodell
  - Ausführung nach LIFO (und damit nicht „quasi-parallel”)
  - CPU und Übersetzer stellen Elementaroperationen bereit
- Koroutinen $\mapsto$ symmetrisches Fortsetzungsmodell
  - Ausführung in beliebiger Reihenfolge
  - erfordert eigenen Kontext: minimal PC, i. a. auch Register und Stapel
  - CPU und Übersetzer stellen i. a. keine Elementaroperationen bereit
- Fäden $\mapsto$ vom BS verwaltete Koroutinen

Referenzen

Melvin E. Conway. „Design of a separable transition-diagram compiler“. In: *Communications of the ACM* 6 (7 Juli 1963), S. 396–408. ISSN: 0001-0782. DOI: 10.1145/366663.366704.

Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997. ISBN: 978-0201896831.

Michael Philippsen. *Grundlagen des Übersetzerbaus*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 2, 2015 (jährlich). URL: https://www2.cs.fau.de/teaching/WS2015/UE1/index.html.

John C. Reynolds. „The discoveries of continuations“. In: *Lisp Symb. Comput.* 6 (3-4 Nov. 1993), S. 233–248. ISSN: 0892-4635. DOI: 10.1007/BF01019459.

Wolfgang Schröder-Preikschat. *Systemprogrammierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_SP.

Betriebssysteme (BS)

VL 9 – Fadenverwaltung

Volkmar Sieh / **Daniel Lohmann**

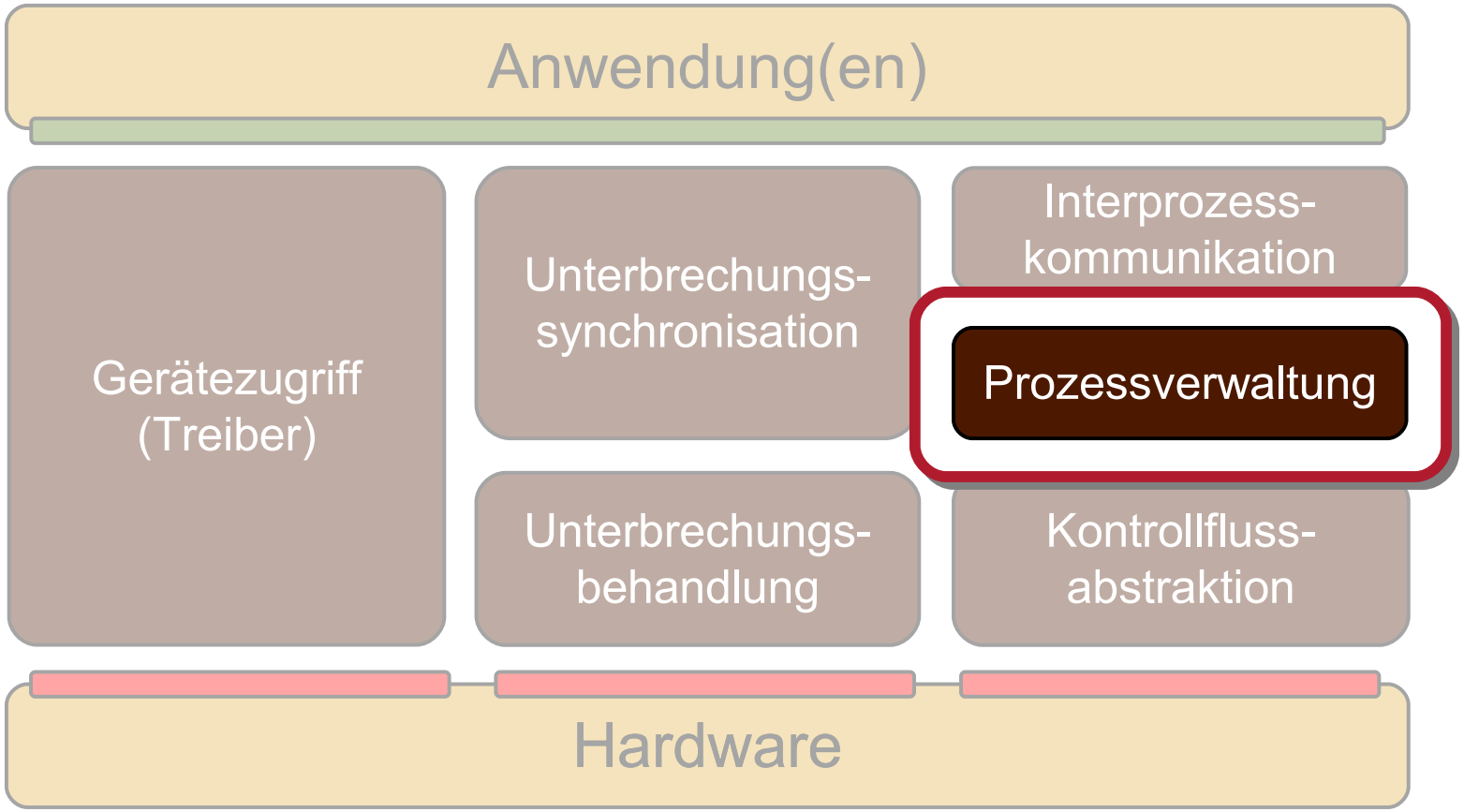
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 20. Dezember 2023

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Agenda

Einordnung
Betriebssystemfäden
Motivation
Kooperativer Fadenwechsel
Präemptiver Fadenwechsel
Ablaufplanung
Zusammenfassung

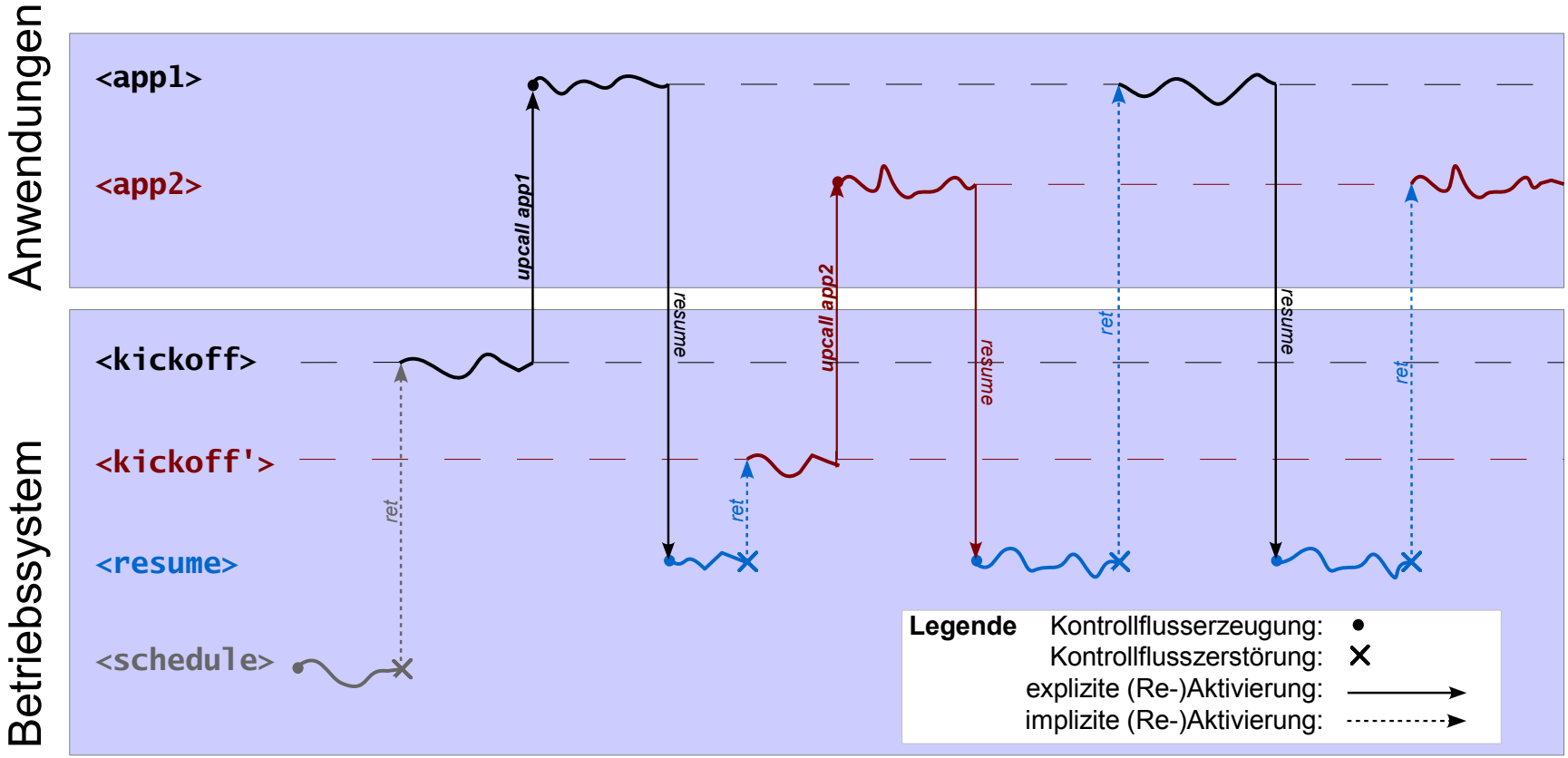
Agenda

Einordnung
Betriebssystemfäden
Motivation
Kooperativer Fadenwechsel
Präemptiver Fadenwechsel
Ablaufplanung
Zusammenfassung

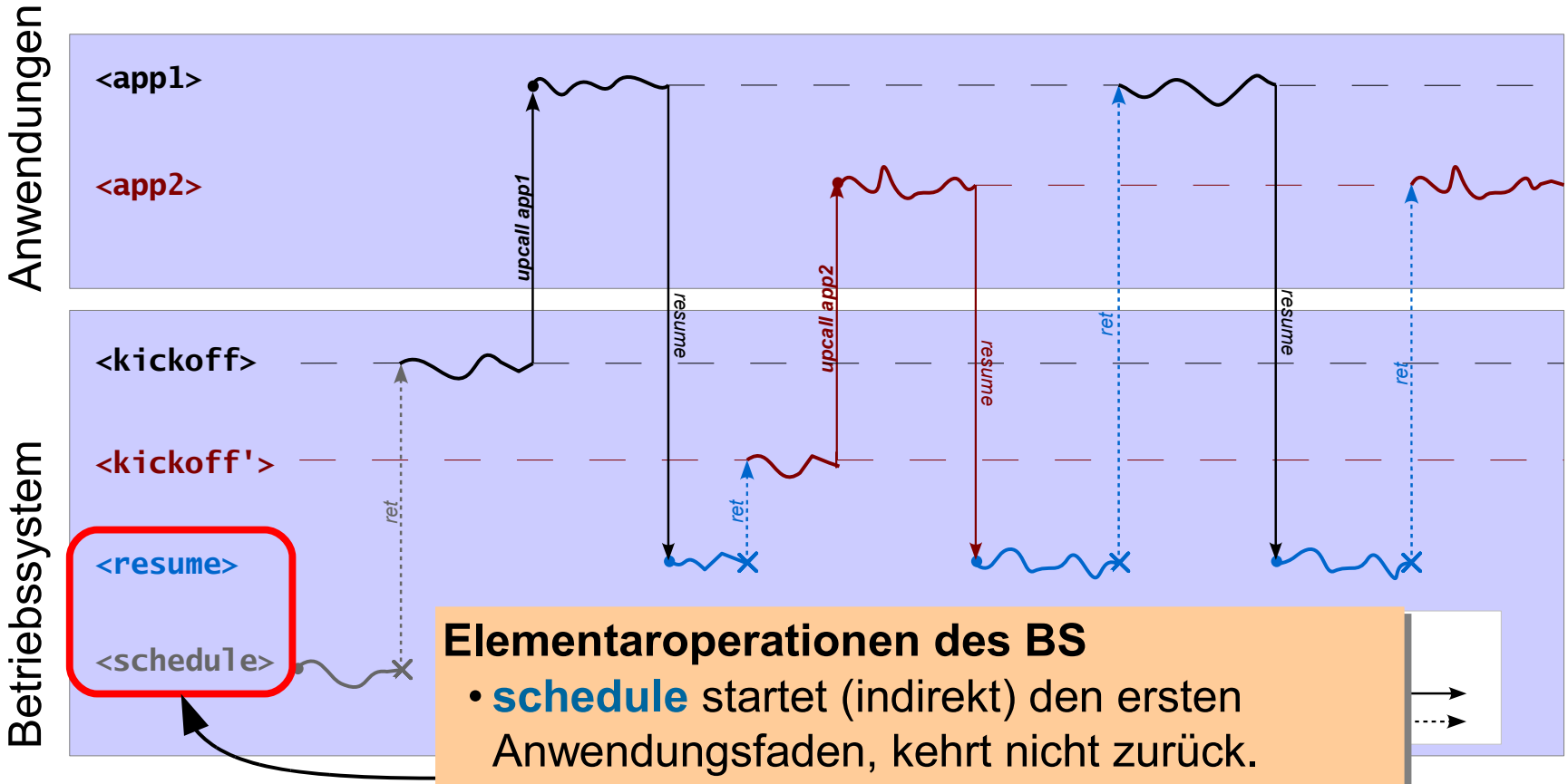
Betriebssystemfäden: Motivation

- Ansatz: Anwendungen „unbemerkt“ als eigenständige Fäden ausführen
  - eine BS-Koroutine pro Anwendung
  - Aktivierung der Anwendung erfolgt durch Aufruf
  - Koroutinenwechsel erfolgt indirekt durch Systemaufruf
- Vorteile
  - unabhängige Anwendungsentwicklung
  - Ablaufplanung (*Scheduling*) wird zentral implementiert
  - bei E/A kann eine Anwendung einfach vom BS „blockiert“ und später wieder „geweckt“ werden
  - zusätzlicher Entzugsmechanismus (*preemption mechanism*) kann die Monopolisierung der CPU verhindern

Kooperativer Fadenwechsel



Kooperativer Fadenwechsel

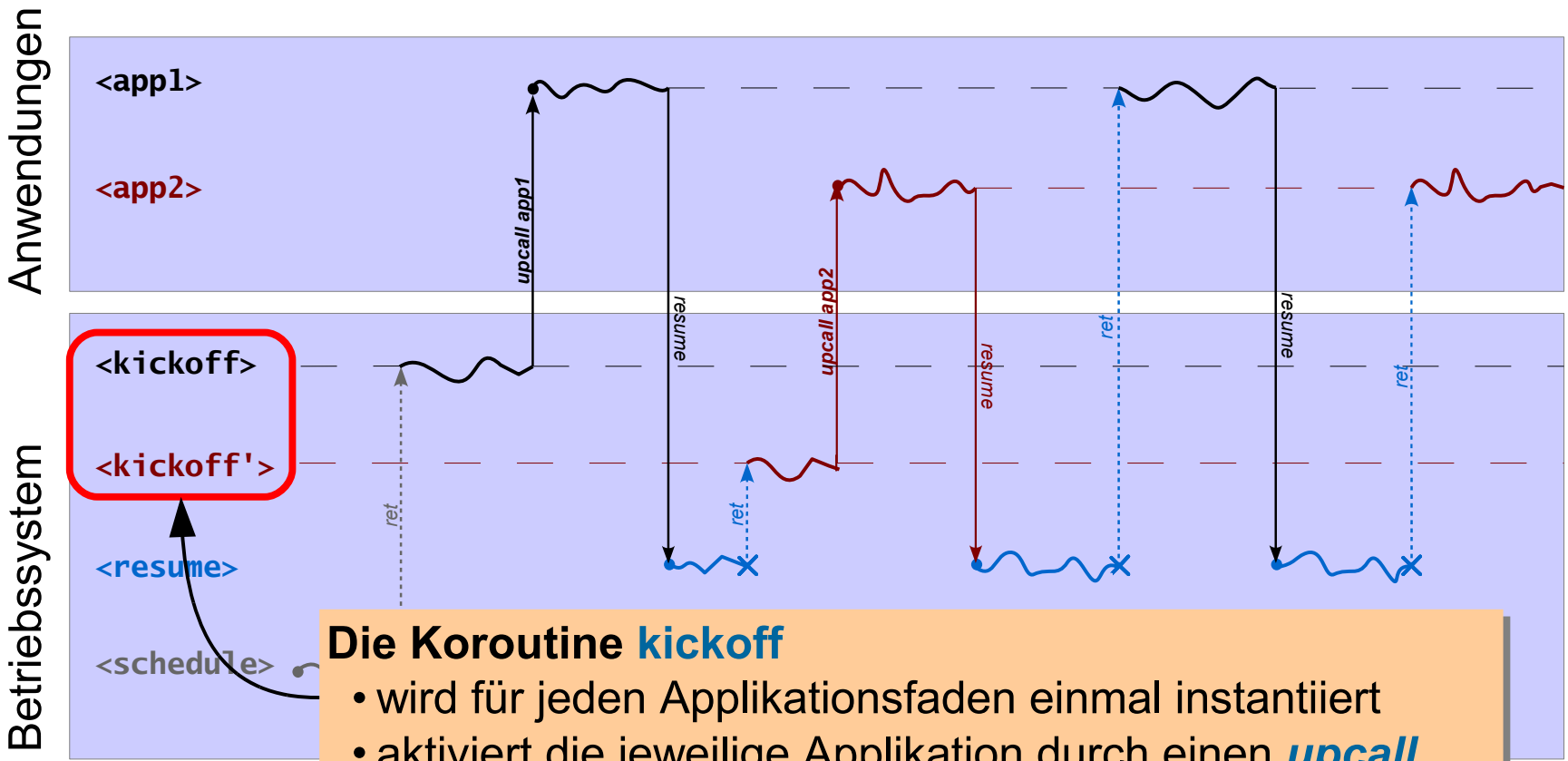


Elementaroperationen des BS

- **schedule** startet (indirekt) den ersten Anwendungsfaden, kehrt nicht zurück.
- **resume** wechselt von einem Anwendungsfaden zum nächsten.

Beide treffen eine **Scheduling-Entscheidung**.

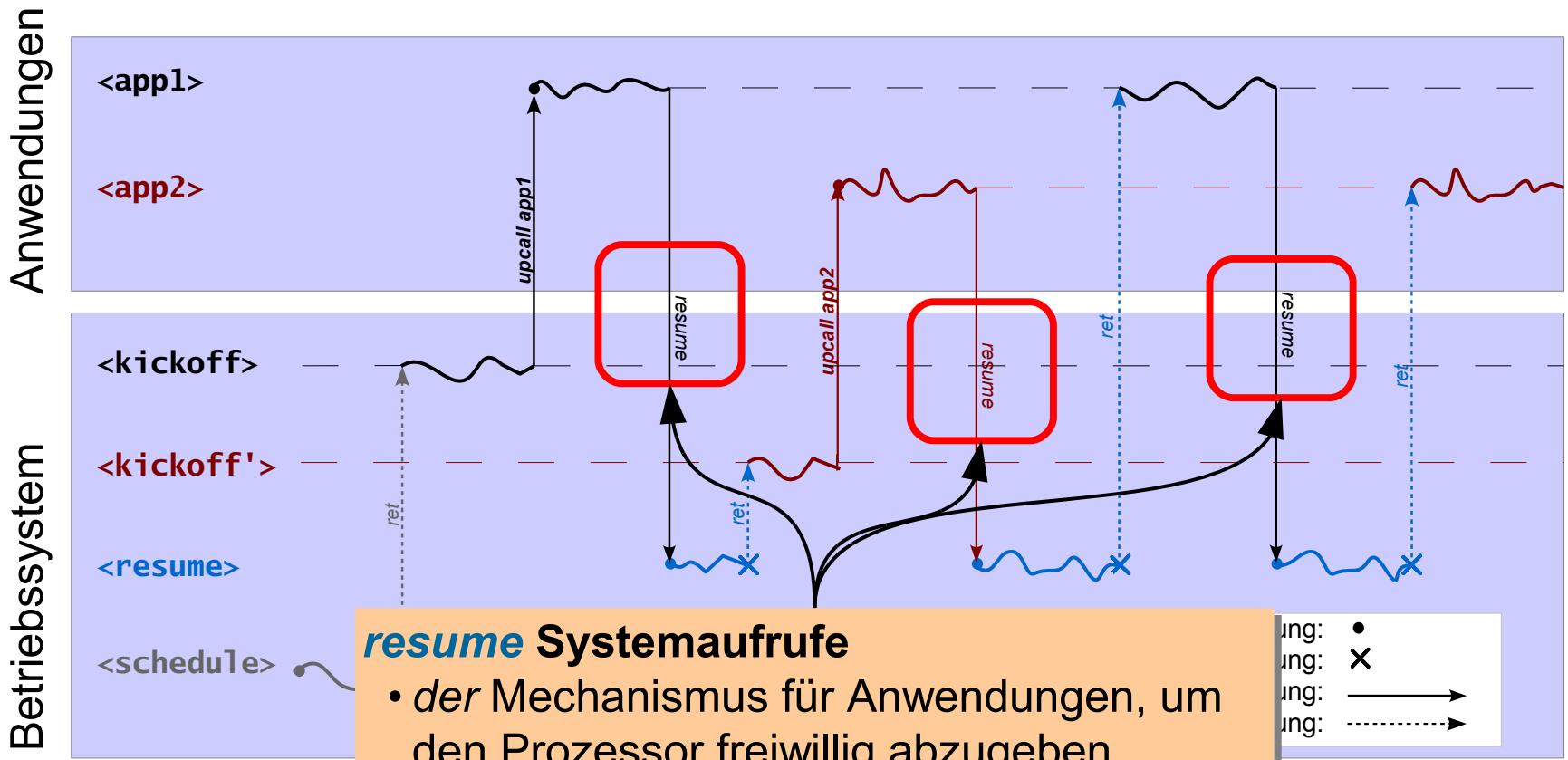
Kooperativer Fadenwechsel



Die Koroutine kickoff

- wird für jeden Applikationsfaden einmal instantiiert
- aktiviert die jeweilige Applikation durch einen ***upcall***

Kooperativer Fadenwechsel

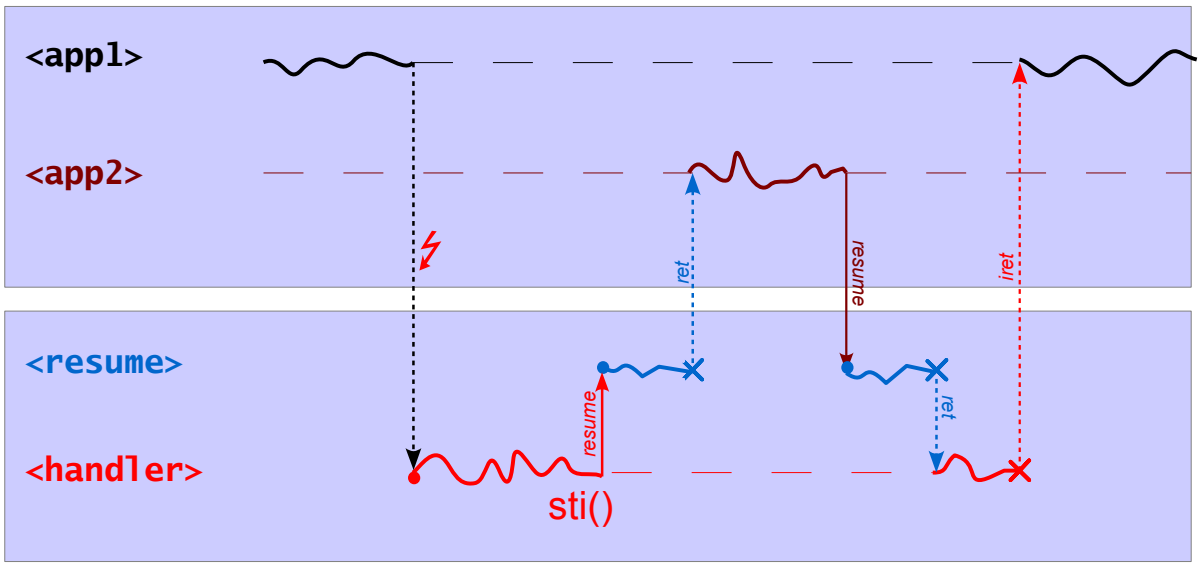


***resume* Systemaufrufe**

- *der* Mechanismus für Anwendungen, um den Prozessor freiwillig abzugeben
- ggf. verbunden mit einem Moduswechsel der CPU (in diesem Fall wird noch ein *Wrapper* benötigt)

Präemptiver Fadenwechsel

- CPU-Entzug durch Zeitgeberunterbrechung
  - die Unterbrechung ist „nur“ ein impliziter Aufruf
  - Behandlungsroutine kann *resume* aufrufen



Achtung: So geht es normalerweise *nicht*, denn *resume* trifft eine **Scheduling-Entscheidung**. Bei den notwendigen Datenstrukturen ist **Unterbrechungssynchronisation** zu beachten!

Agenda

Einordnung
Betriebssystemfäden
Motivation
Kooperativer Fadenwechsel
Präemptiver Fadenwechsel
Ablaufplanung
- Grundbegriffe und Klassifizierung
- unter Windows
- unter Linux

Zusammenfassung

Arbeitsteilung

- ***Scheduler***
  - trifft **strategische Entscheidungen** zur Ablaufplanung
  - betrachtet wird immer eine Menge lauffähiger Fäden
    - die Fäden sind allgemein in einer CPU-Warteschlange aufgereiht
    - die Sortierung erfolgt entsprechend der **Scheduling-Strategie**
  - laufende Faden ist immer von der Entscheidung mit betroffen
    - dazu muss der laufende Faden jederzeit „greifbar“ sein
    - vor der Umschaltung ist ist der laufende Faden zu vermerken
  - ein ausgewählter neuer Faden wird dem *Dispatcher* übergeben
- ***Dispatcher***
  - setzt die Entscheidungen durch und schaltet Fäden (mit Hilfe von *resume*) um

Ablaufplanung: Einteilung ...

- nach der **Betriebsmittelart**
  der zeitweilig belegten Hardware-Resourcen
- nach der **Betriebsart**
  des zu bedienenden/steuernden Rechnersystems
- nach dem **Zeitpunkt**
  der Erstellung des Ablaufplans
- nach der **Vorhersagbarkeit**
  von Zeitpunkt und Dauer von Prozessabläufen
- nach dem **Kooperationsverhalten**
  der (Benutzer/System-) Programme
- nach der **Rechnerarchitektur**
  des Systems
- nach der **Ebene der Entscheidungsfindung**
  bei der Betriebsmittelvergabe

... nach der Betriebsmittelart

- ***CPU scheduling***
  des Betriebsmittels ”CPU“
  - die Prozessanzahl zu einem Zeitpunkt ist höher als die Prozessoranzahl
  - ein Prozessor ist zwischen mehreren Prozessen zu multiplexen
  - Prozesse werden dem Prozessor über eine Warteschlange zugeteilt
- ***I/O scheduling***
  des Betriebsmittels ”Gerät“, speziell: ”Platte“
  - gerätespezifische Einplanung der von Prozessen abgesetzten E/A-Aufträge
  - *disk scheduling*, z.B., berücksichtigt typischerweise drei Faktoren:
    – (1) Positionszeit, (2) Rotationszeit, (3) Transferzeit
  - Geräteparameter und Gerätezustand bestimmen die nächste E/A-Aktion
  - die getroffenen Entscheidungen sind ggf. nicht konform zum *CPU scheduling*

... nach der Betriebsart

- ***batch scheduling***
  interaktionsloser bzw. unabhängiger Programme
  - nicht-verdrängende bzw. verdrängende Verfahren mit langen Zeitscheiben
  - Minimierung der Kontextwechselanzahl
- ***interactive scheduling***
  interaktionsreicher bzw. abhängiger Programme
  - ereignisgesteuerte, verdrängende Verfahren mit kurzen Zeitscheiben
  - Antwortzeitminimierung durch Optimierung der Systemaufrufe
- ***real-time scheduling***
  zeitkritischer bzw. abhängiger Programme
  - ereignis- oder zeitgesteuerte **deterministische** Verfahren
  - Garantie der Einhaltung umgebungsbedingter Zeitvorgaben
  - Rechtzeitigkeit ist entscheidend und nicht Geschwindigkeit

... nach dem Zeitpunkt

- ***online scheduling***
  dynamisch, **während** der eigentlichen Ausführung
  - interaktive und Stapelsysteme, aber auch weiche Echtzeitsysteme

- ***offline scheduling***
  statisch, **vor** der eigentlichen Ausführung
  - wenn die Komplexität eine Ablaufplanung im laufenden Betrieb verbietet
    - Einhaltung aller Zeitvorgaben garantieren: ein NP-vollständiges Problem
    - kritisch, wenn auf jede abfangbare katastrophale Situation zu reagieren ist
  - Ergebnis der Vorberechung ist ein vollständiger Ablaufplan (in Tabellenform)
    - (semi-) automatisch erstellt per Quelltextanalyse spezieller ”Übersetzer“
    - oft zeitgesteuert abgearbeitet/ausgeführt als Teil der Prozessabfertigung
  - die Verfahren sind zumeist beschränkt auf strikte Echtzeitsysteme

... nach der Vorhersagbarkeit

- ***deterministic scheduling***
  bekannter, exakt vorberechneter Prozesse
  - Prozesslaufzeiten/-termine sind bekannt, sie wurden ggf. ”offline“ berechnet
  - die genaue Vorhersage der CPU-Auslastung ist möglich
  - das System garantiert die Einhaltung der Prozesslaufzeiten/-termine
  - die Zeitgarantien gelten unabhängig von der jeweiligen Systemlast
- ***probabilistic scheduling***
  unbekannter Prozesse
  - Prozesslaufzeiten/-termine bleiben unbestimmt
  - die (wahrscheinliche) CPU-Auslastung kann lediglich abgeschätzt werden
  - das System kann Zeitgarantien nicht geben und auch nicht einhalten
  - Zeitgarantien sind durch Anwendungsmaßnahmen bedingt erreichbar

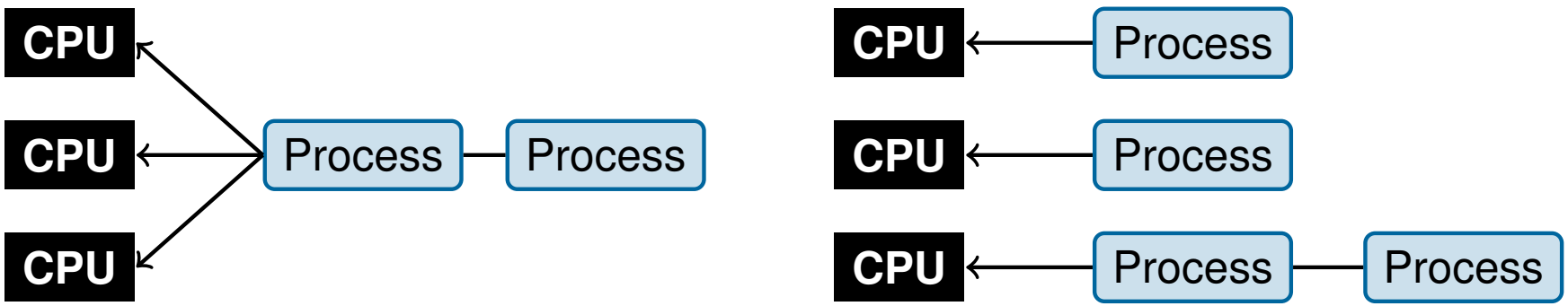
... nach dem Kooperationsverhalten

- ***cooperative scheduling***
  voneinander abhängiger Prozesse
  - Prozesse müssen die CPU freiwillig abgeben, zugunsten anderer Prozesse
  - die Programmausführung muss (direkt/indirekt) **Systemaufrufe** bewirken
  - die Systemaufufe müssen (direkt/indirekt) den *Scheduler* aktivieren

- ***preemptive scheduling***
  voneinander unabhängiger Prozesse
  - Prozessen wird die CPU entzogen, zugunsten anderer Prozesse
  - **Ereignisse** können die Verdrängung des laufenden Prozesses bewirken
  - die Ereignisverarbeitung aktiviert (direkt/indirekt) den *Scheduler*

... nach der Rechnerarchitektur

- ***uni-processor scheduling***
  in Mehr{programm,prozess}systemen
  - die Verarbeitung von Prozessen kann nur pseudo-parallel erfolgen
- ***multi-processor scheduling***
  in Systemen mit gemeinsamen Speicher
  - die parallele Verarbeitung von Prozessen wird ermöglicht
    - alle Prozessoren arbeiten eine globale Warteschlange ab
    - jeder Prozessor arbeitet seine lokale Warteschlange ab



... nach der Ebene

- ***long-term scheduling*** [s – min]
  kontrolliert den Grad an Mehrprogrammbetrieb
  - Benutzer Systemzugang gewähren, Programme zur Ausführung zulassen
  - Prozesse dem *medium*- bzw. *short-term scheduling* zuführen
- ***medium-term scheduling*** [ms – s]
  als Teil der Ein-/Auslagerungsfunktion
  - Programme zwischen Vorder- und Hintergrundspeicher hin- und herbewegen
  - *swapping*: auslagern (*swap-out*), einlagern (*swap-in*)
- ***short-term scheduling*** [µs – ms]
  regelt die Prozessorzuteilung an die Prozesse
  - ereignisgesteuerte Ablaufplanung: Unterbrechungen, Systemaufrufe, Signale
  - Blockierung bzw. Verdrängung des laufenden Prozesses

Scheduling-Kriterien

- **Antwortzeit** Minimierung der Zeitdauer von der Auslösung einer Systemanforderung bis zur Entgegennahme der Rückantwort, bei gleichzeitiger Maximierung der Anzahl interaktiver Prozesse.
- **Durchlaufzeit** Minimierung der Zeitdauer vom Starten eines Prozesses bis zu seiner Beendigung, d.h., der effektiven Prozesslaufzeit und aller Prozesswartezeiten.
- **Termineinhaltung** Starten und/oder Beendigung eines Prozesses zu einem fest vorgegebenen Zeitpunkt.
- **Vorhersagbarkeit** Deterministische Ausführung des Prozesses unabhängig von der jeweils vorliegenden Systemlast.
- **Durchsatz** Maximierung der Anzahl vollendeter Prozesse pro vorgegebener Zeiteinheit. Liefert ein Maß für die geleistete Arbeit im System.
- **Prozessorauslastung** Maximierung des Prozentanteils der Zeit, während der die CPU Prozesse ausführt, d.h., ”sinnvolle“ Arbeit leistet.
- **Gerechtigkeit** Gleichbehandlung der auszuführenden Prozesse und Zusicherung, den Prozessen innerhalb gewisser Zeiträume die CPU zuzuteilen.
- **Dringlichkeiten** Bevorzugte Verarbeitung des Prozesses mit der höchsten (statisch/dynamisch zugeordneten) Priorität.
- **Lastausgleich** Gleichmäßige Betriebsmittelauslastung bzw. bevorzugte Verarbeitung der Prozesse, die stark belastete Betriebsmittel eher selten belegen.

Scheduling-Kriterien

- **Antwortzeit**
- **Durchlaufzeit**
- **Termineinhaltung**
- **Vorhersagbarkeit**
- **Durchsatz**
- **Prozessorauslastung**
- **Gerechtigkeit**
- **Dringlichkeiten**
- **Lastausgleich**

Benutzerorientierte Kriterien
- wahrgenommenes Systemverhalten
- bestimmen die Akzeptanz durch Benutzer

beeinflussen

Systemorientierte Kriterien
- effiziente Nutzung der Betriebsmittel
- bestimmen die Kosten des Rechnerbetriebs

Kriterien bei typischen Betriebsarten

- allgemein (unabhängig von der Betriebsart)
  - Gerechtigkeit
  - Lastausgleich
- **Stapelsysteme**
  - Durchsatz
  - Durchlaufzeit
  - Prozessorauslastung
- **interaktive Systeme**
  - Antwortzeit
    (Proportionalität – Bearbeitungsdauer entspricht Erwartung)
- **Echtzeitsysteme**
  - Dringlichkeit
  - Termineinhaltung
  - Vorhersagbarkeit

Agenda

Einordnung
Betriebssystemfäden
Motivation
Kooperativer Fadenwechsel
Präemptiver Fadenwechsel
Ablaufplanung
- Grundbegriffe und Klassifizierung
- unter Windows
- unter Linux

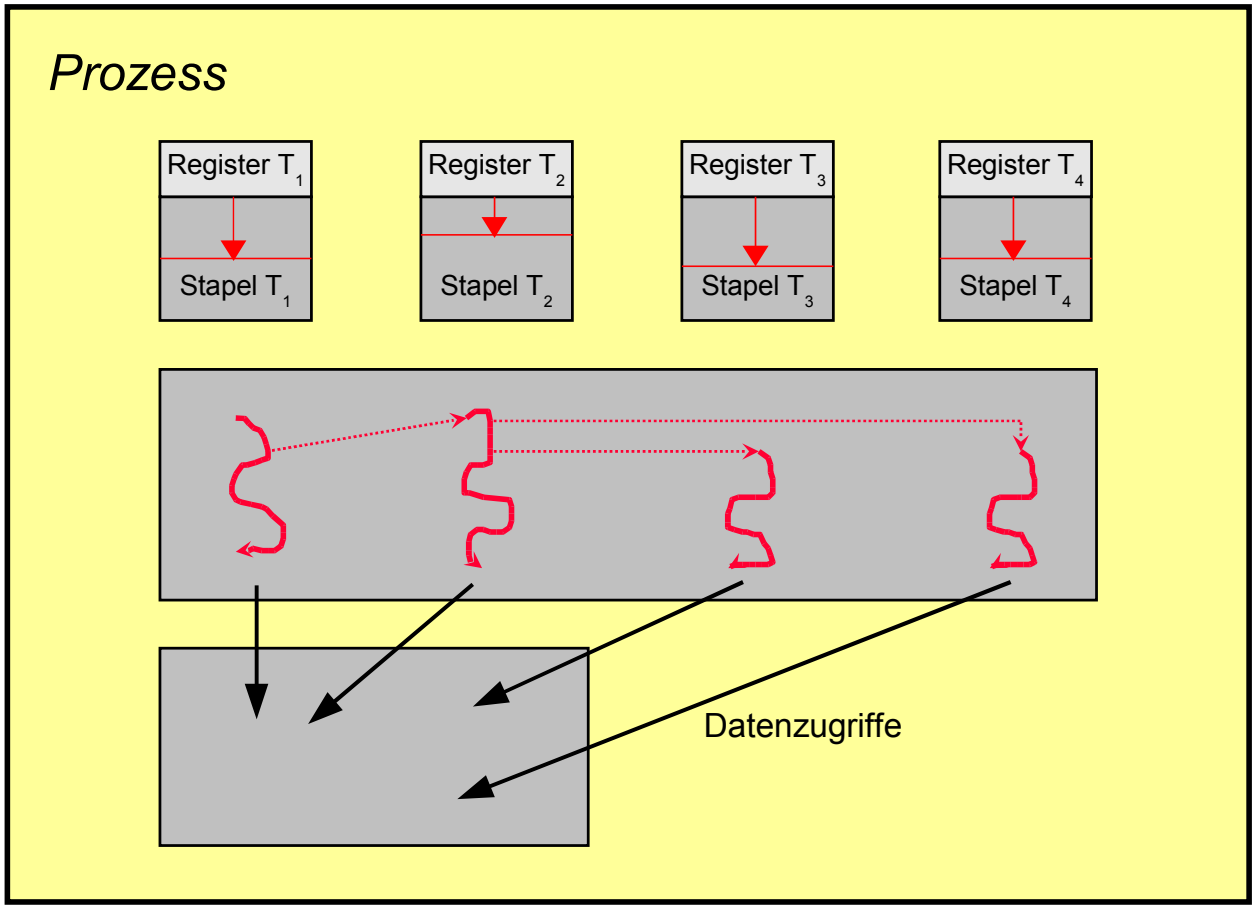
Zusammenfassung

Prozesse und Fäden in Windows

Stapel + Registersatz (1 je Faden)

Code

Gobale und statische Daten



Prozesse und Fäden in Windows

- Prozess: Umgebung und Adressraum für Fäden
  - Ein Win32 Prozess enthält immer mindestens einen Faden
  - Faden (engl. thread): Code ausführende Einheit
- Fadenimplementierung wird durch den NT Systemkern erbracht
  - *Usermode-Threads* möglich (*„Fibers“*), aber unüblich
- *„Threads“* bekommen vom *Scheduler* Rechenzeit zugeteilt

Der Windows-Scheduler

- Preemptives, prioritätengesteuertes Scheduling:
  - Thread mit höherer Priorität verdrängt Thread niedrigerer Priorität
    - Egal ob Thread sich im User- oder Kernelmode befindet
    - Die meisten Funktionen der Executive („Kernel“) sind ebenfalls als Threads implementiert
  - Round-Robin bei Threads gleicher Priorität
    - Zuteilung erfolgt reihum für eine Zeitscheibe (Quantum)
- Thread-Prioritäten
  - Derzeit 0 bis 31, aufgeteilt in drei Bereiche
    - Variable Priorities: 1 bis 15
    - Realtime Priorities: 16 bis 31
    - Priorität 0 ist reserviert für den Nullseiten-Thread
  - Threads der Executive verwenden maximal Priorität 23

Zeitscheiben (Quantum)

	Kurze Quantumwerte		Lange Quantumwerte	
	Variabel	**Fix**	**Variabel**	**Fix**
Thread in HG-Prozess	6	18	12	36
Thread in VG-Prozess	12	18	24	36
Aktiver Thread in VG-Prozess	18	18	36	36

- Quantum wird vermindert
  - um den Wert 3 bei jedem Clock-Tick (alle 10 bzw. 15 msec)
  - um den Wert 1, falls Thread in den Wartezustand geht
- Länge einer Zeitscheibe: 20 – 180 msec

Prioritätsklassen, relative Threadpriorität

Relative Thread Priority		Process Priority Class: Idle	Below Normal	Normal	Above Normal	High	Realtime
		4	6	8	10	13	24
Time Critical	=15	15	15	15	15	15	31
Highest	+2	6	8	10	12	15	26
Above Normal	+1	5	7	9	11	14	25
Normal		4	6	8	10	13	24
Below Normal	-1	3	5	7	9	12	23
Lowest	-2	2	4	6	8	11	22
Idle	=1	1	1	1	1	1	16

Prioritäten: *Variable Priorities*

- *Variable Priorities* (1-15)
  - Scheduler verwendet Strategien, um „wichtige“ Threads zu bevorzugen
    - *Quantum-Stretching* (Bevorzugung des aktiven GUI-Threads)
    - dynamische Anhebung (*Boost*) der Priorität für wenige Zeitscheiben bei Ereignissen
  - Fortschrittsgarantie
    - Alle 3 bis 4 Sekunden bekommen bis zu 10 „benachteiligte“ Threads für zwei Zeitscheiben die Priorität 15
  - Threadpriorität berechnet sich wie folgt (vereinfacht):

    Prozessprioritätsklasse + Threadpriorität + Boost

Prioritäten: *Realtime Priorities*

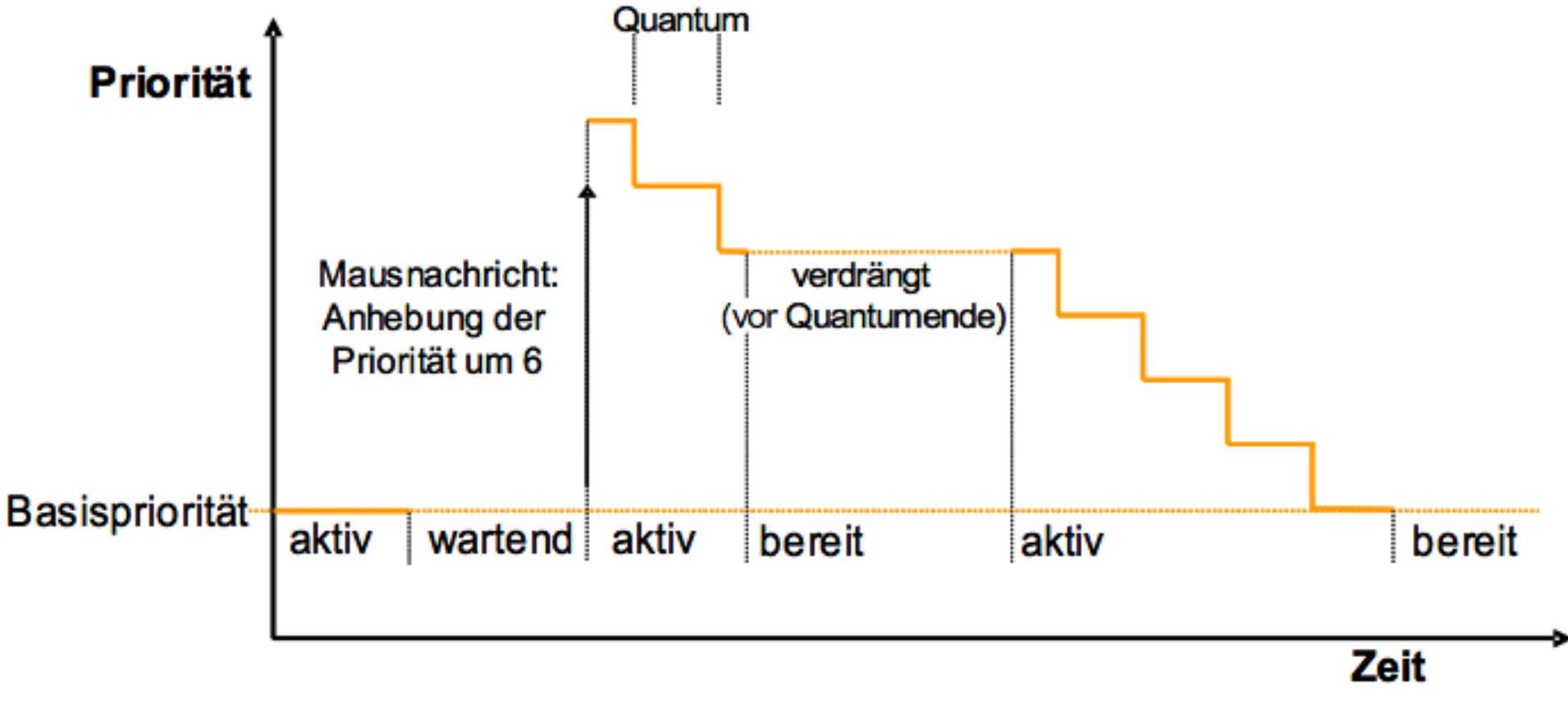
- *Realtime Priorities* (16-31)
  - Reines prioritätengesteuertes Round-Robin
    - Keine Fortschrittsgarantie
    - Keine dynamische Anhebung
    - Betriebssystem kann negativ beeinflusst werden
    - Spezielles Benutzerrecht erforderlich (SeIncreaseBasePriorityPrivilege)
  - Threadpriorität berechnet sich wie folgt:

    REALTIME_PRIORITY_CLASS + Threadpriorität

Dynamische Prioritätsanpassung

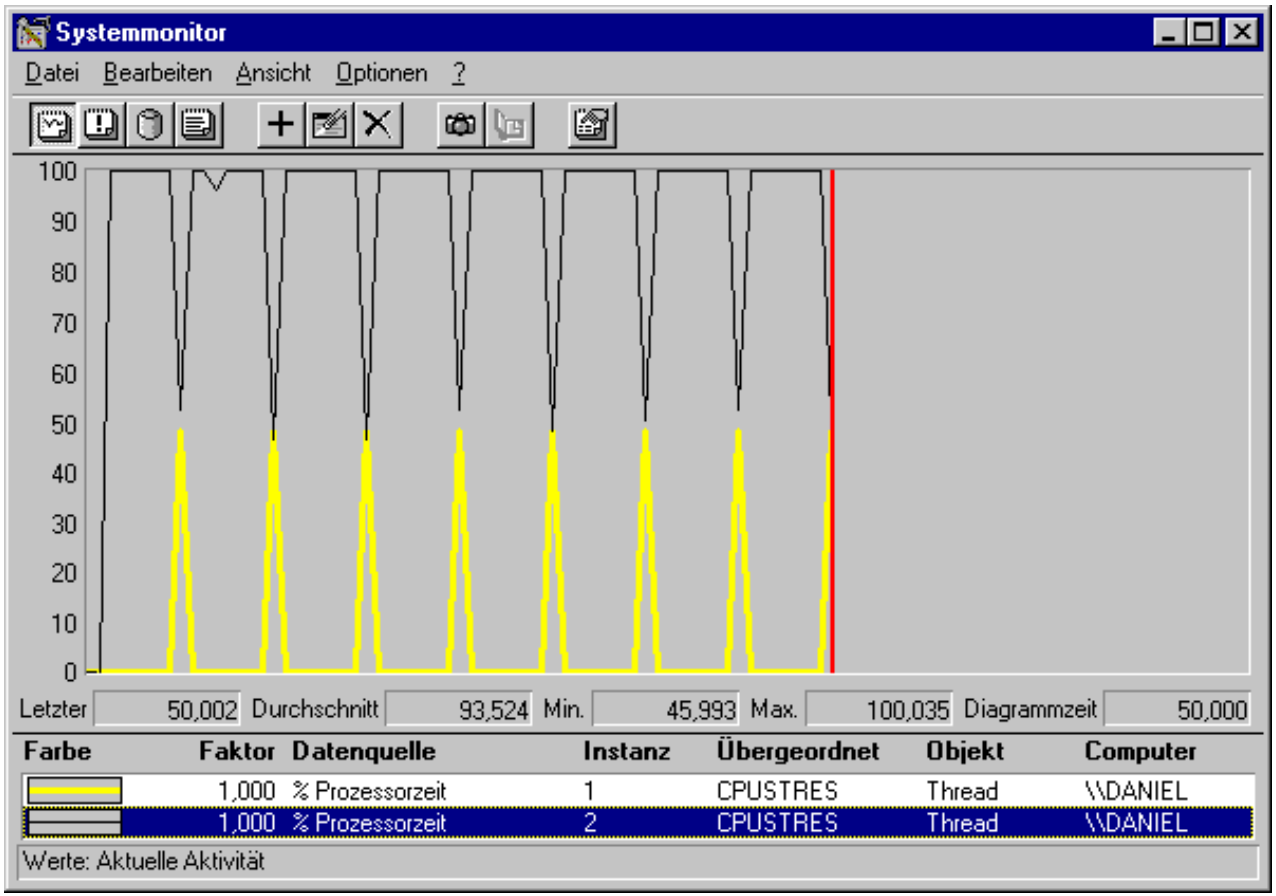
- *Dynamic Boosts*
  - Thread-Prioritäten werden vom System in bestimmten Situationen dynamisch angehoben (nicht bei REALTIME_PRIORITY_CLASS)
    - Plattenein/ausgabe abgeschlossen: +1
    - Maus-, Tastatureingabe: +6
    - Semaphore, Event, Mutex: +1
    - Andere Ereignisse (Netzwerk, Pipe,...) +2
    - Ereignis in Vordergrundapplikation +2
  - Dynamic Boost wird „verbraucht“ (eine Stufe pro Quantum)

Prioritätänderung nach einem Boost



Der *Balance-Set-Manager*

- Etwa alle 3-4 Sekunden erhalten bis zu 10 „benachteiligte“ Threads für zwei Zeitscheiben die Priorität 15
  - Implementierung der Fortschrittsgarantie



Auswahl des nächsten Threads (SMP)

Ziel: „gerechtes“ RoundRobin bei max. Durchsatz

Problem: Cache-Effekte

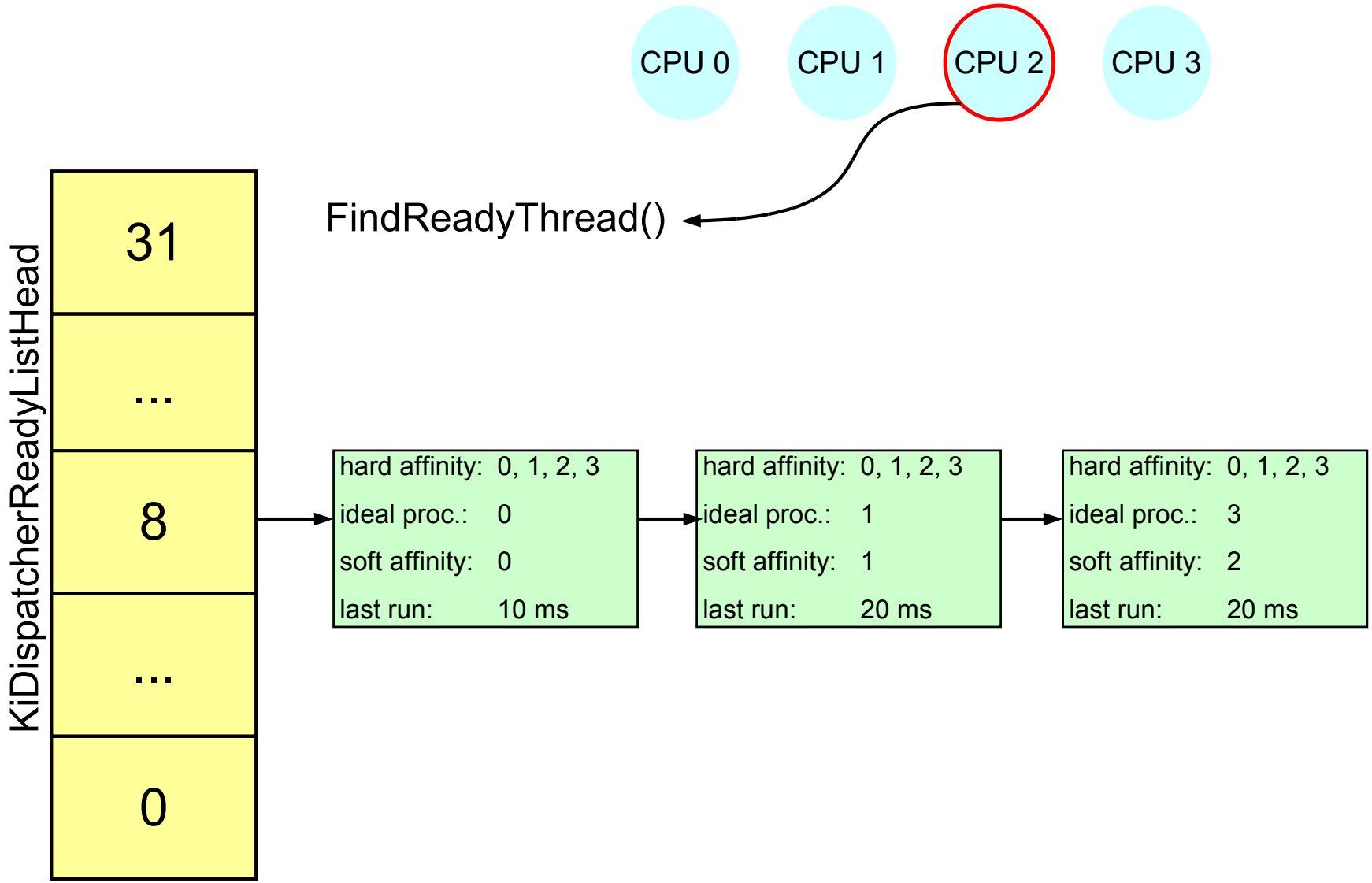
Affinität (Zuordnung von CPUs zu Thread):

- hard_affinity: Feste Zuordnung
  → explizit durch SetThreadAffinity() zugewiesen
- ideal_processor: „Ideale“ Zuordnung
  → implizit bei Erzeugung zugewiesen („zufällig“)
  → änderbar mit SetThreadIdealProcessor()
- soft_affinity: Letzte CPU, auf welcher der Thread lief
  → intern vom Scheduler verwaltet
- last_run: Zeitpunkt der letzten Zuweisung zu einer CPU
  → intern vom Scheduler verwaltet

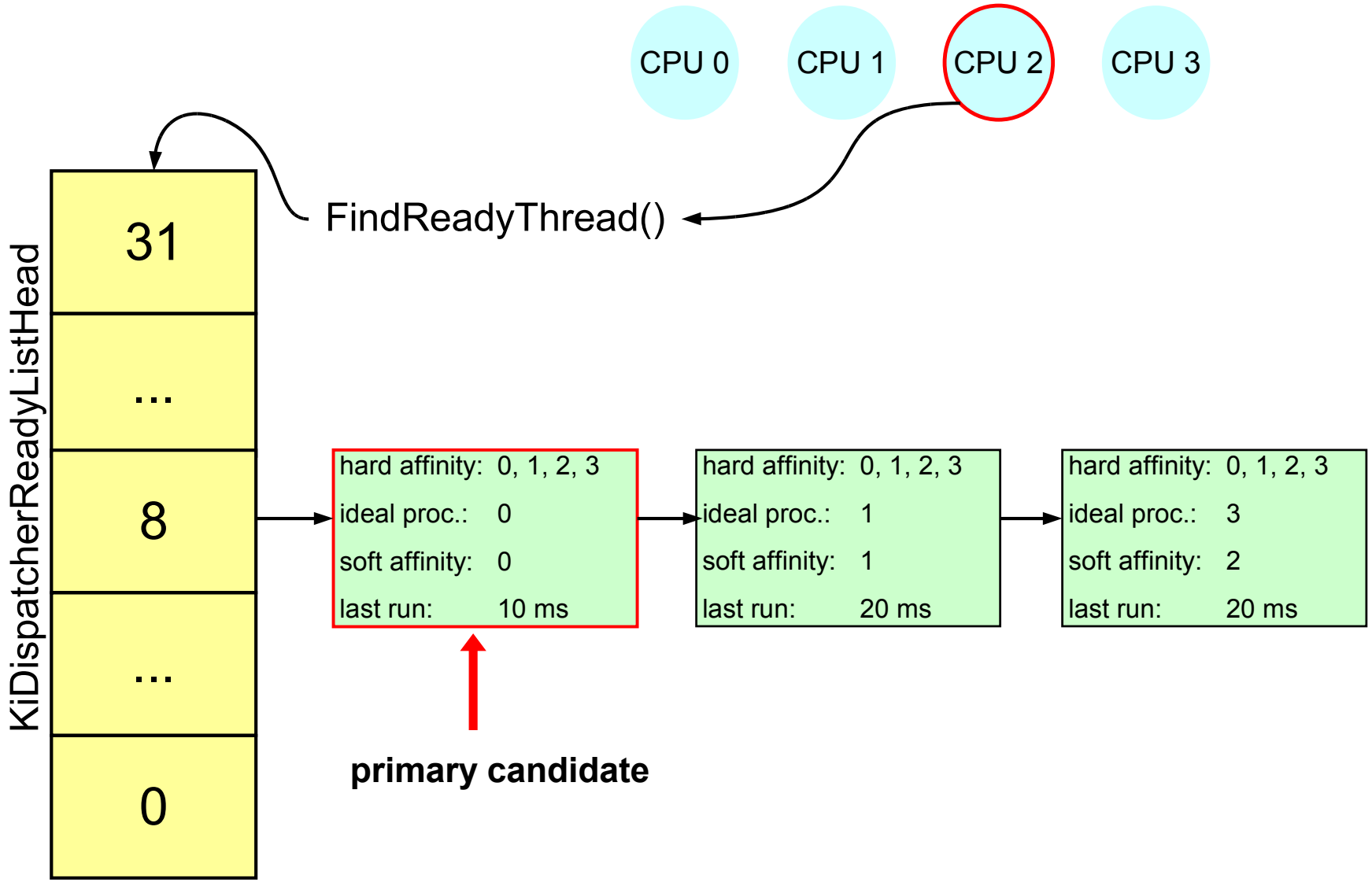
Auswahl des nächsten Threads (SMP)

- Algorithmus: CPU *n* ruft FindReadyThread() auf
  - Wähle höchstpriore, nicht-leere Warteschlange
  - Suche in dieser Warteschlange nach Thread, mit
    - soft_affinity == *n* oder
    - ideal_processor == *n* oder
    - currentTime() – last_run > 2 Quantum oder
    - priority >= 24
  - Sonst wähle Kopf der Warteschlange

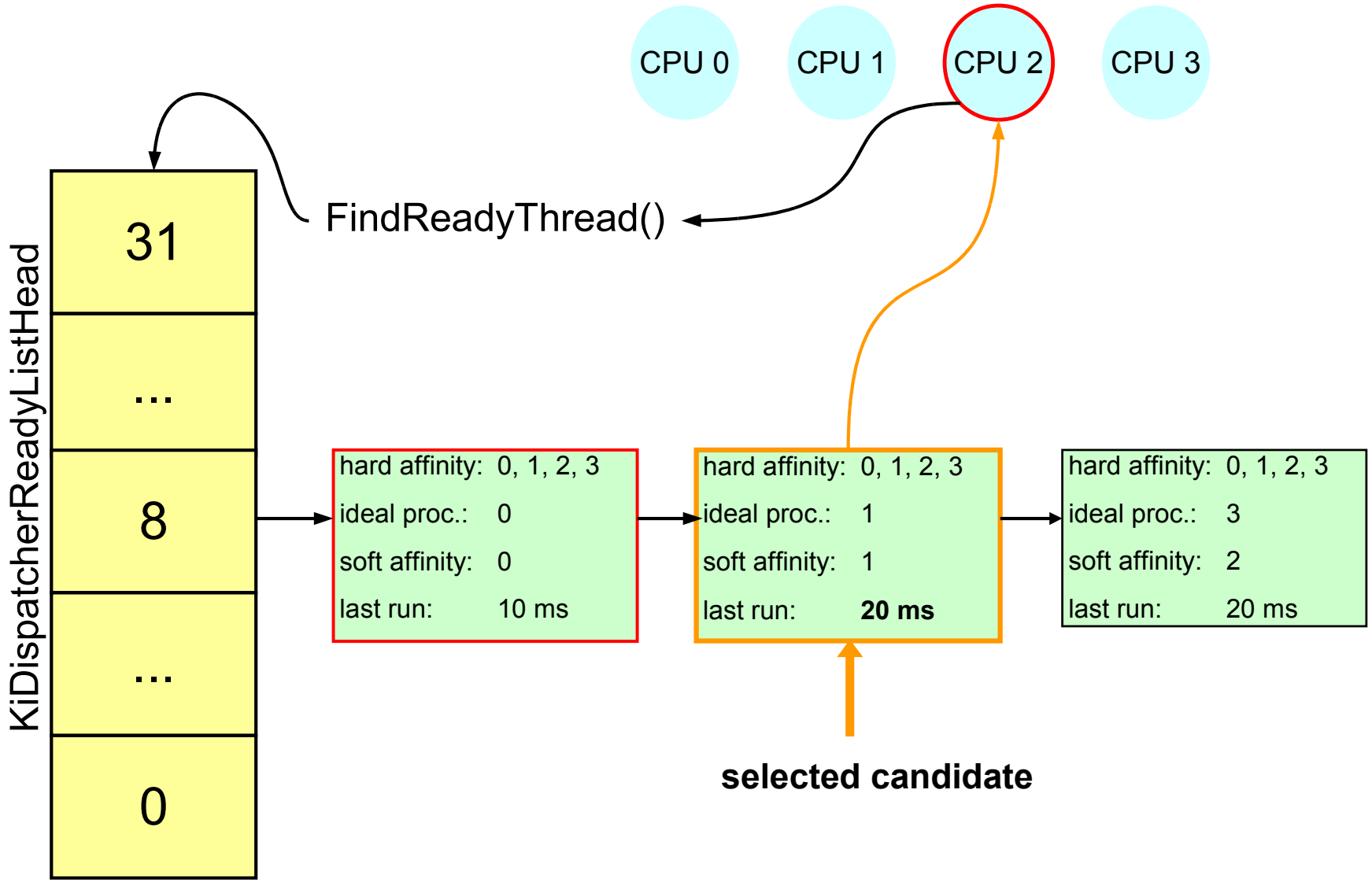
Auswahl des nächsten Threads (SMP)



Auswahl des nächsten Threads (SMP)



Auswahl des nächsten Threads (SMP)



Änderungen ab Windows 2003

- Eine ReadyQueue pro CPU
- Algorithmus: CPU n ruft FindReadyThread() auf
  - Wähle höchstpriore, nicht-leere Warteschlange von CPU n
  - Wähle Kopf dieser Warteschlange
  - Falls ReadyQueue komplett leer ist, aktiviere Idle-Loop
  - Im Idle-Loop: Durchsuche ReadyQueue anderer CPUs

Fazit Windows

- ***„interactive, probabilistic, online, preemptive, multiprocessor CPU scheduling“***
- Prioritätenmodell erlaubt feine Zuteilung der Prozessorzeit
  - Dynamische Anpassungen beachten
  - Usermode-Threads mit hohen Echtzeitprioritäten haben Vorrang vor allen System-Threads!
  - Executive ist im allgemeinen unterbrechbar
- Interaktive Threads können bevorzugt werden
  - Insbesondere GUI/Multimedia-zentrierte Threads
- Weitere Verbesserungen für SMP in Windows 2003

Agenda

Einordnung
Betriebssystemfäden
Motivation
Kooperativer Fadenwechsel
Präemptiver Fadenwechsel
Ablaufplanung
- Grundbegriffe und Klassifizierung
- unter Windows
- unter Linux

Zusammenfassung

Linux Scheduler: Historie

- Kernel < 2.2: Ring-Warteschlange mit Round Robin
  - kein SMP Support
  - keine Echtzeitprioritäten
- Kernel ≥ 2.2: Erster SMP-Support
  - Einführung von *scheduling classes*
- Kernel ≥ 2.4: O(n) Scheduler
  - Mehr Fairness durch Scheduler-Epochen
  - Miese Leistung auf SMP-Systemen
- Kernel ≥ 2.5: O(1) Scheduler
  - Effiziente Epochenverwaltung durch *active* und *expired* Listen
  - Per-Core *runqueues* mit Lastausgleich
  - Viele Heuristiken zur Bevorzugung interaktiver Anwendungen
- Kernel ≥ 2.6.23: *Completely Fair Scheduler (CFS)*
  - Schlanker Scheduler, der die Rechenzeit "ideal fair" verteilt
  - Hierarchisches Scheduling durch *scheduler groups*

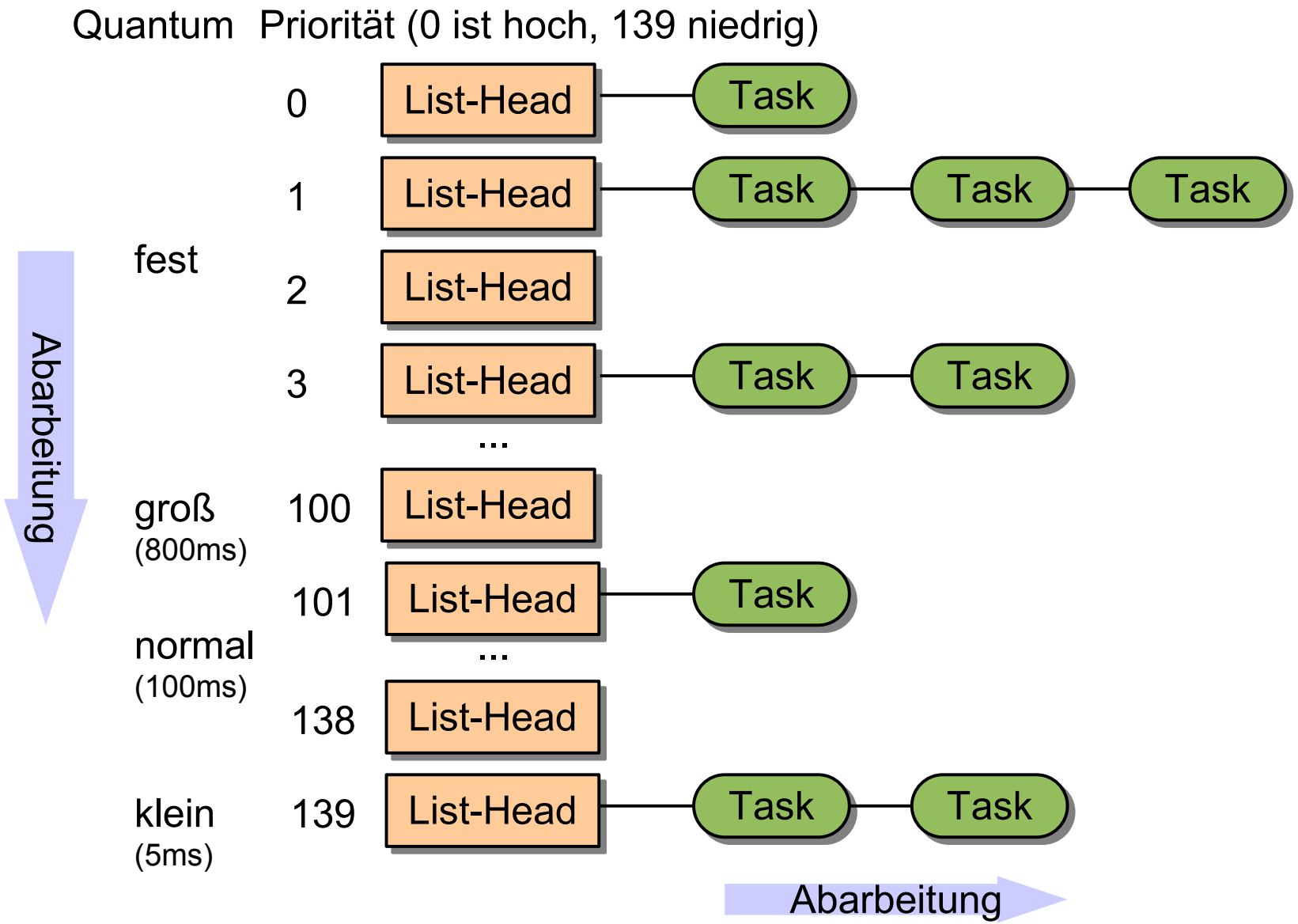
Linux *Tasks* ...

- sind die **Linux *Kernel*-Abstraktion** für ...
  - **UNIX Prozesse**: ein Kontrollfaden in einem Adressraum
  - **Linux *Threads***: spezieller Prozess, der sich seinen virtuellen Adressraum mit mindestens einem anderen *Thread* teilt
- sind die vom Scheduler betrachteten Aktivitäten

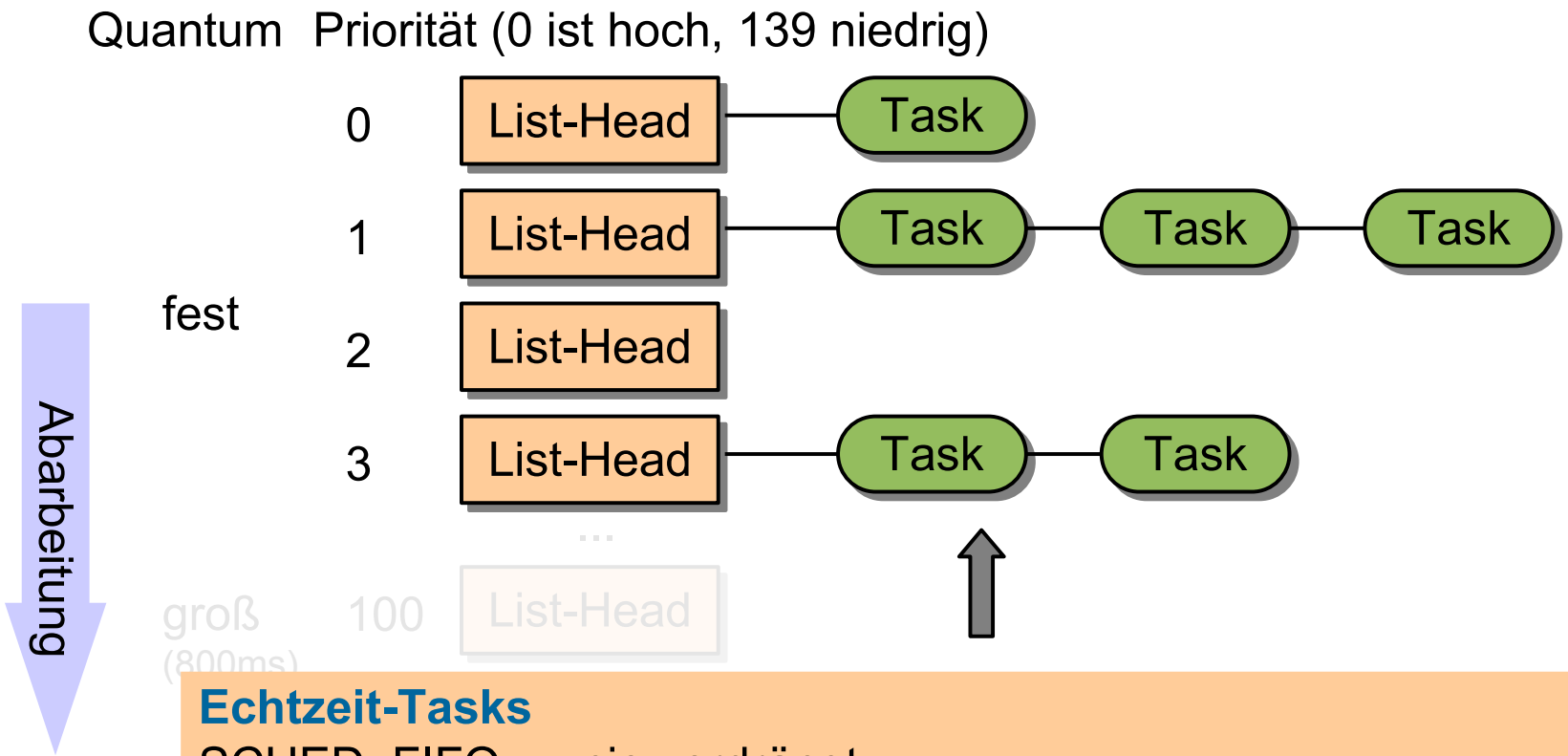
➔ ein Programm mit vielen Threads bekommt unter Linux mehr Rechenzeit als **ein** klassischer Prozess

- gleiches gilt allerdings auch für ein Programm mit einem Prozess und vielen Kindprozessen

Multi-Level Queues



Multi-Level Queues



Echtzeit-Tasks

SCHED_FIFO nie verdrängt

SCHED_RR verdrängt, wenn feste Zeitscheibe abgelaufen

Echtzeit-Tasks verdrängen jeden anderen normalen Task.

Durch die einfache Strategie kann das Verhalten in einer weichen Echtzeitumgebung recht gut vorhergesagt werden.

Abarbeitung

O(1) Scheduler: Multi-Level Queues

Quantum Priorität (0 ist hoch, 139 niedrig)

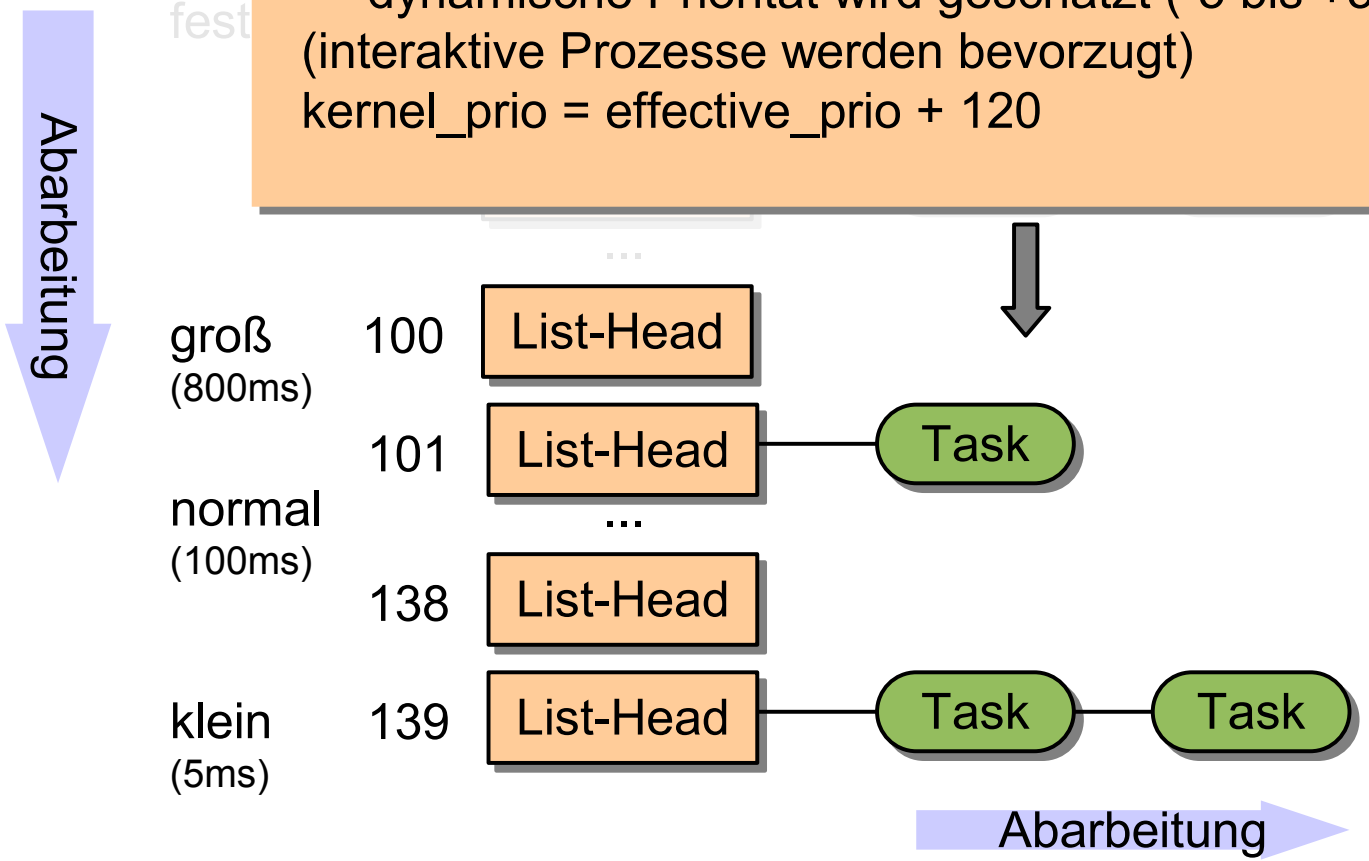
Gewöhnliche Tasks

effective_prio = static_prio + dynamic_prio

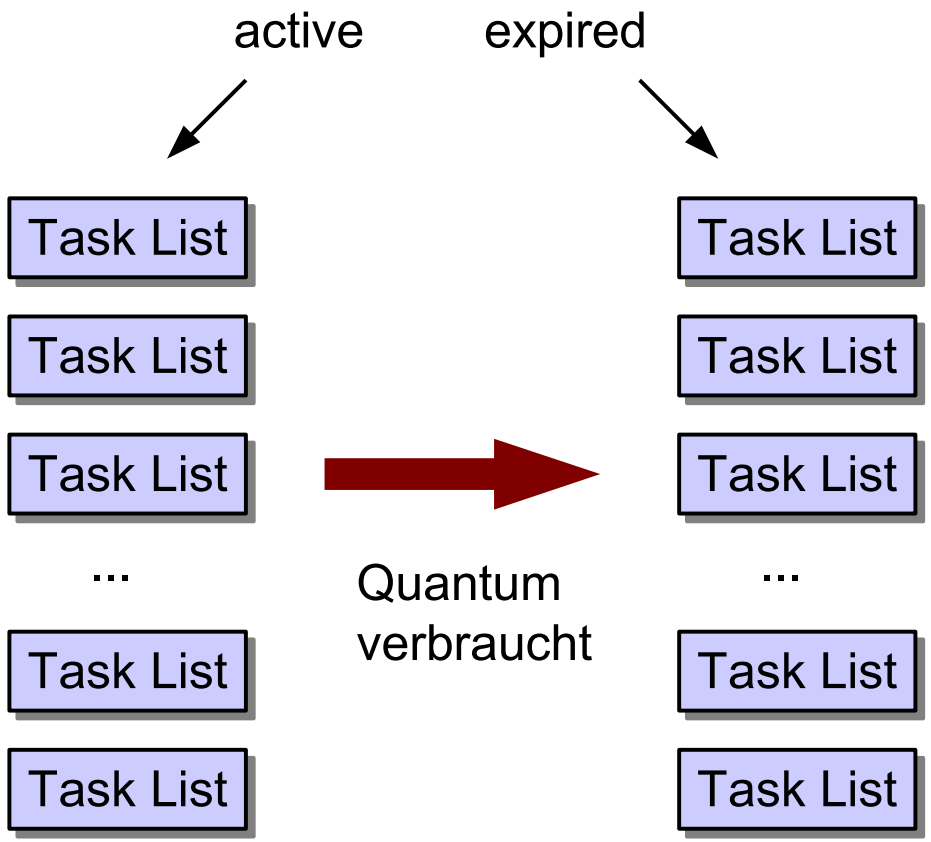
- statische Priorität entspricht dem nice value (-19 bis +20)
- dynamische Priorität wird geschätzt (-5 bis +5)

(interaktive Prozesse werden bevorzugt)

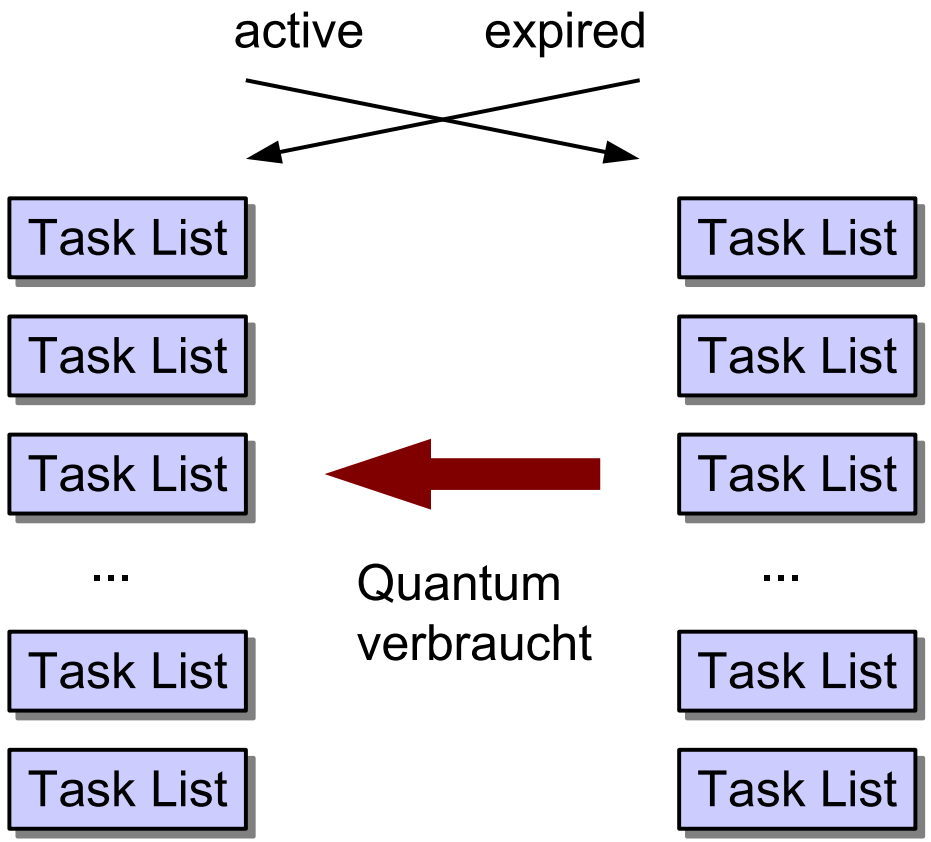
kernel_prio = effective_prio + 120



O(1) Scheduler: Active und Expired



O(1) Scheduler: Active und Expired



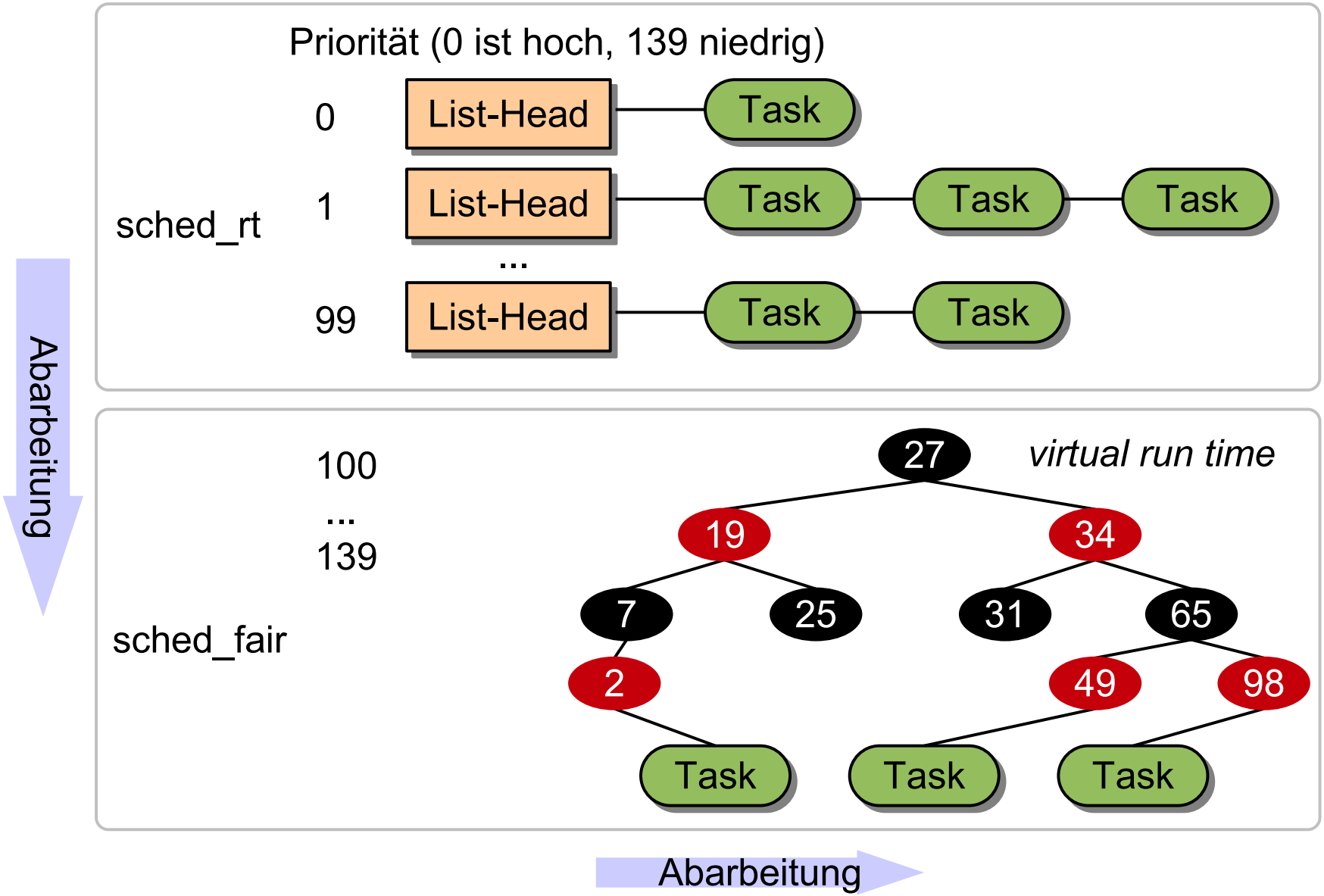
Fazit Linux O(1) Scheduler

- ***„interactive, probabilistic, online, preemptive, multiprocessor CPU scheduling“***
- Bevorzugung interaktiver Prozesse
  - schnelle Reaktion auf Eingaben
  - gleichzeitig Fortschrittgarantie für CPU-lastige Prozesse
- O(1) bei allen Operationen des Scheduler
  - Einfügen, Entfernen, Scheduling-Entscheidung
- Mehrprozessorunterstützung
  - Mehrere Bereit-Listen: Parallele Scheduler Ausführung
  - Keine Idle-Phasen (war ein Problem beim alten Linux Scheduler)
  - CPU-Lastausgleich

Completely Fair Scheduler (CFS)

- Ansatz: Ablaufbereite Tasks bekommen die Rechenzeit gleichmäßig ("fair") zugeteilt
  - bei *n* Tasks jeweils 1/*n*-tel der CPU-Leistung
  - hierarchische Zuteilung durch *scheduling groups*
- CFS läuft nur bei SCHED_NORMAL
  - Echtzeittask (SCHED_RR und SCHED_FIFO) wie bisher
  - ansonsten: Task mit *geringster* CPU-Zeit hat höchste Priorität
- Scheduling-Kriterium ist die bislang zugeteilte CPU-Zeit
  - Ready-Liste als Rot-Schwarz-Baum, sortiert nach der Zeit
  - Komplexität O(log *N*)
    (in der Praxis trotzdem effizienter als alter O(1)-Scheduler)
  - Prioritäten (im Sinne von nice) werden durch "schnellere/langsamere" Uhren abgebildet

Completely Fair Scheduler (CFS)



Fazit: Completly Fair Scheduler (CFS)

- ***„interactive, probabilistic, online, preemptive, multiprocessor CPU scheduling“***
- Keine direkte Bevorzugung interaktiver Prozesse
  - Fortschrittsgarantie für alle Prozesse
  - Im Vergleich zum O(1)-Scheduler: Besserstellung von Hintergrundprozessen
- O(log *n*) bei den meisten Operationen des Scheduler
  - Einfügen, Scheduling-Entscheidung
- Hierarchie durch *scheduling groups*
  - z.B. Fairness zwischen Benutzern: Eine Gruppe pro Benutzer
  - innerhalb der Gruppe: Fairness zwischen allen Mitgliedern
  - Benutzer kann weitere Gruppen erstellen, um "seinen Anteil" aufzuteilen

Agenda

Einordnung
Betriebssystemfäden
Motivation
Kooperativer Fadenwechsel
Präemptiver Fadenwechsel
Ablaufplanung
Zusammenfassung

Zusammenfassung

- *Threads* sind Koroutinen des Betriebssystems
  - BS hat Entzugsmechanismen
  - Strategie der Ablaufplanung wird als *Scheduling* bezeichnet
- *Scheduling* hat großen Einfluss auf die Performanz des Gesamtsystems, es legt fest, ...
  - welche Prozesse warten und welche voranschreiten
  - welche Betriebsmittel wie ausgelastet sind
- Es gibt verschiedenste Varianten des Scheduling
  - nur wenige Unterschiede bei gängigen PC/Workstation-Bestriebsystemen
  - eventuell aber starke Unterschiede in anderen Anwendungsdomänen

Betriebssysteme (BS)

VL 10 – Architekturen

Volkmar Sieh / **Daniel Lohmann**

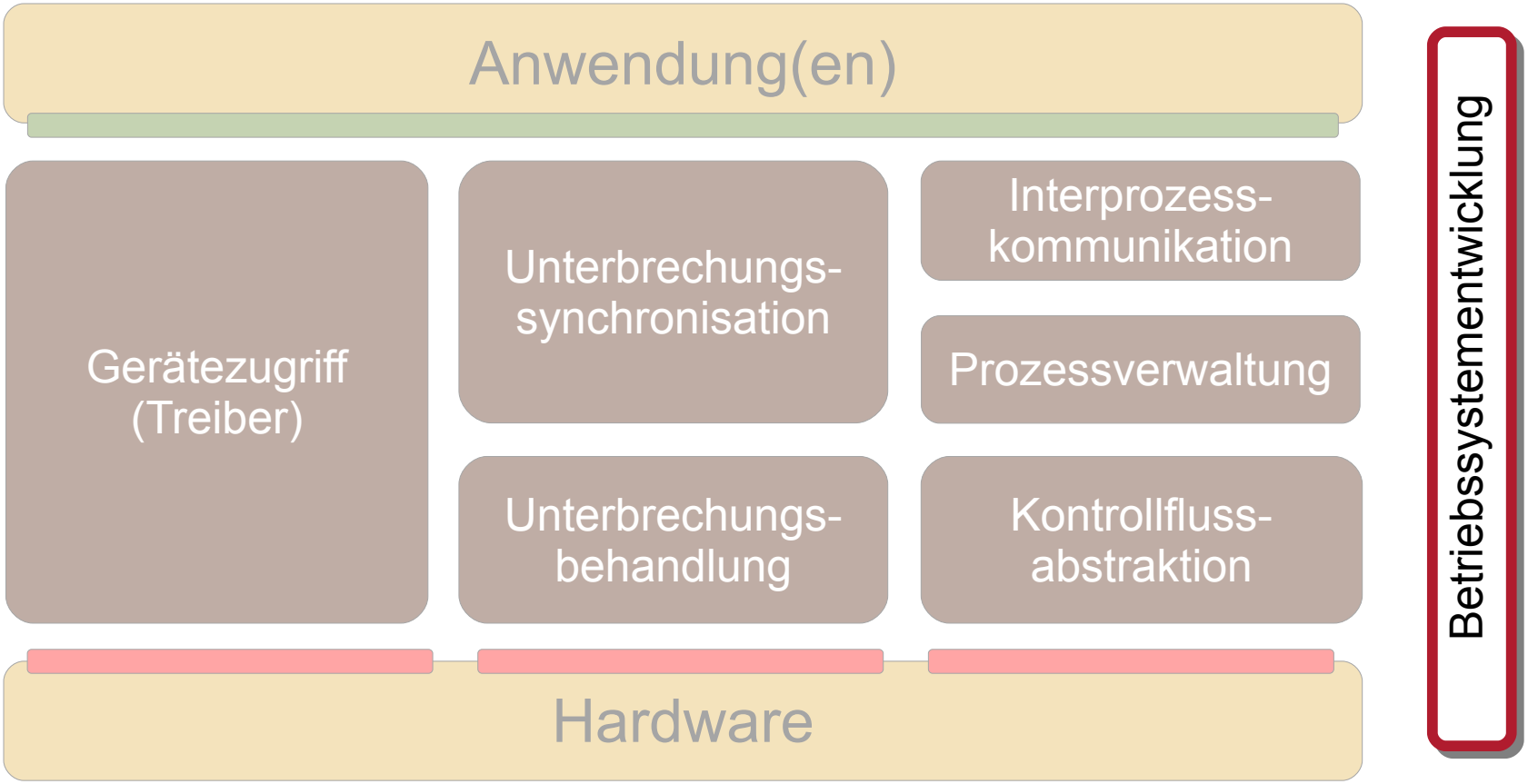
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 10. Januar 2024

`https://sys.cs.fau.de/lehre/ws23/bs`

Überblick: Einordnung dieser VL



Agenda

Einführung
Geschichte, Mode und Trend
Zusammenfassung
Referenzen

Agenda

Einführung
- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung

Geschichte, Mode und Trend

Zusammenfassung

Referenzen

Bewertungskriterien für Betriebssysteme

- Anwendungsorientierte Kriterien
  - **Portabilität**
    - *Wie unabhängig ist man von der Hardware?*
  - **Erweiterbarkeit**
    - *Wie leicht lässt sich das System erweitern (z. B. um neue Gerätetreiber)?*
  - **Robustheit**
    - *Wie stark wirken sich Fehler in Einzelteilen auf das Gesamtsystem aus?*
  - **Leistung**
    - *Wie gut ist die Hardware durch die Anwendung auslastbar?*
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus**
    - *Wie werden Anwendungen / BS-Komponenten isoliert?*
  - **Interaktionsmechanismus**
    - *Wie kommunizieren Anwendungen / BS-Komponenten miteinander?*
  - **Unterbrechungsmechanismus**
    - *Wie werden Unterbrechungen zugestellt und bearbeitet?*

Betriebssystemgeschichte

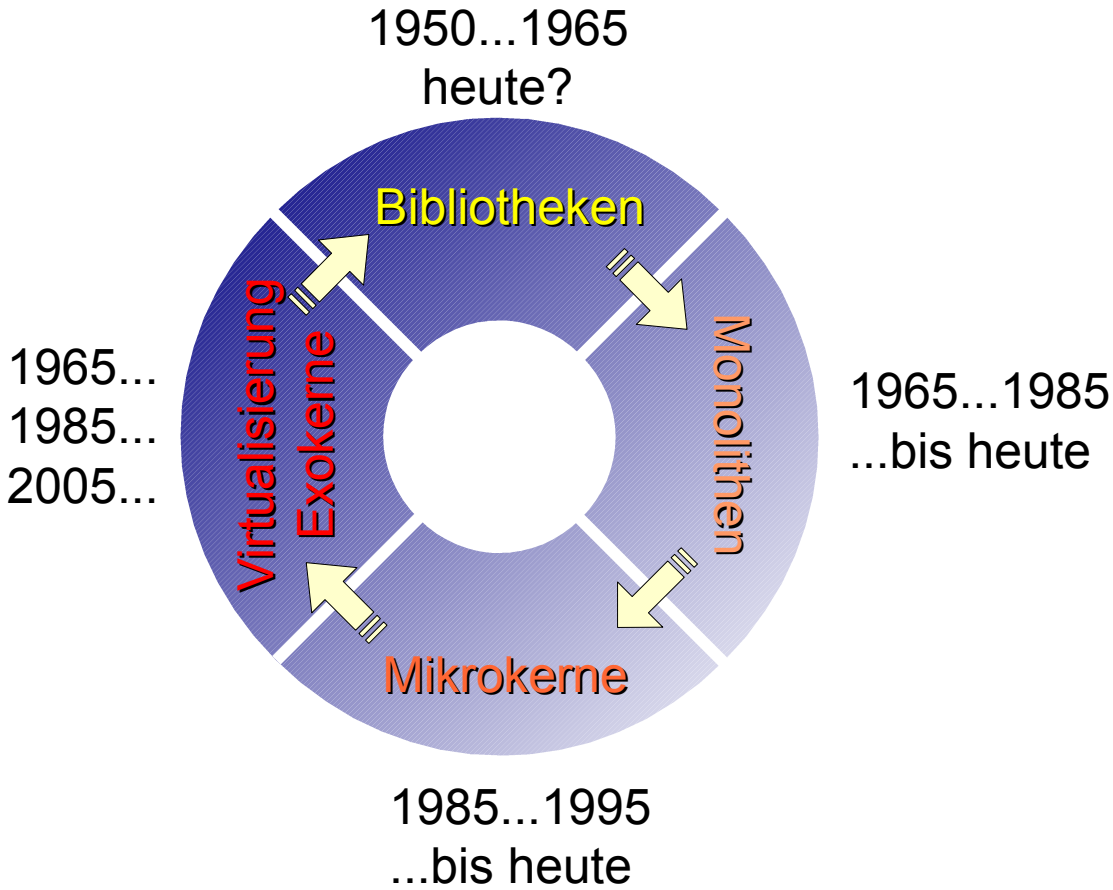
Paradigmen der Betriebssystementwicklung

Definition: Paradigma

Das Wort **Paradigma** [. . .] bedeutet *„Beispiel”*, *„Vorbild”*, *„Muster”* oder *„Abgrenzung”*, *„Vorurteil”*; in allgemeinerer Form auch **„Weltsicht”** oder **„Weltanschauung”**. [Wikipedia]

Betriebssystemgeschichte

Paradigmen der Betriebssystementwicklung



Agenda

Einführung

Geschichte, Mode und Trend

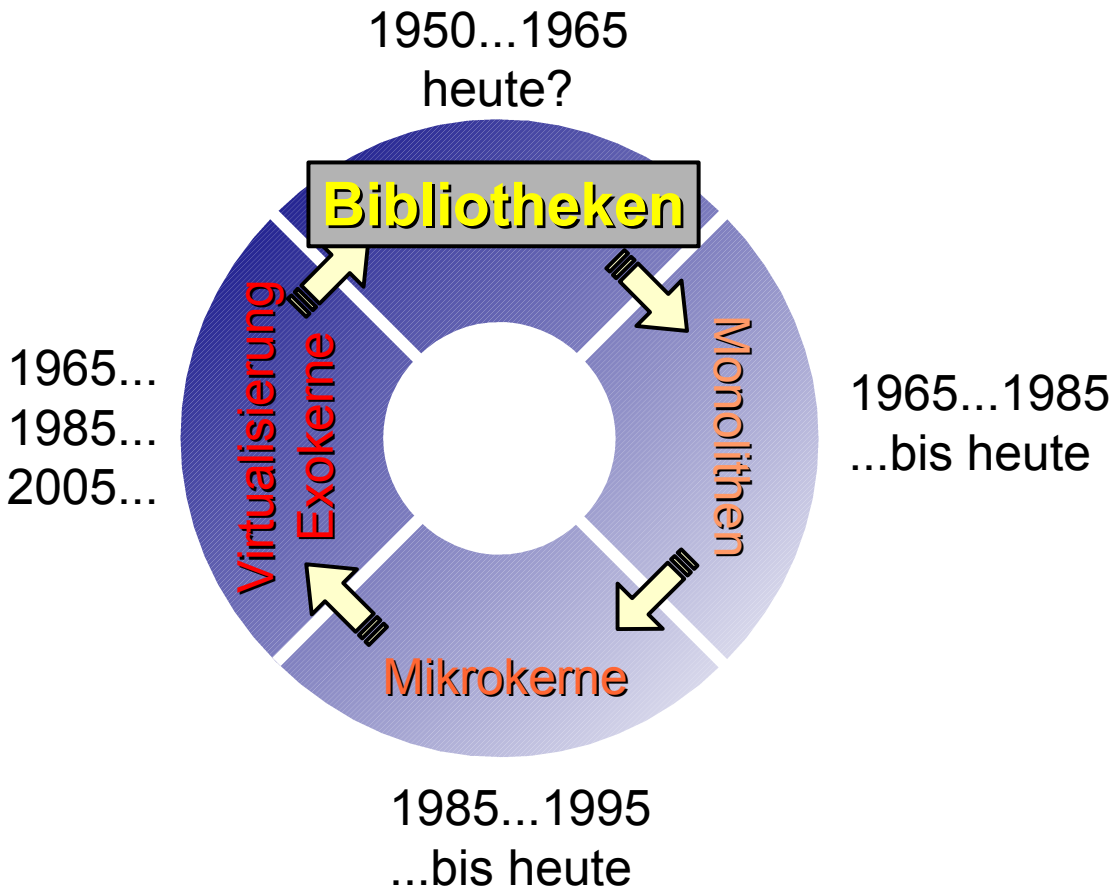
- Bibliotheks-Betriebssysteme
- Monolithen
- Mikrokerne
- Exokerne und Virtualisierung

Zusammenfassung

Referenzen

Überblick: Paradigmen

Funktionsbibliotheken als einfache Infrastrukturen



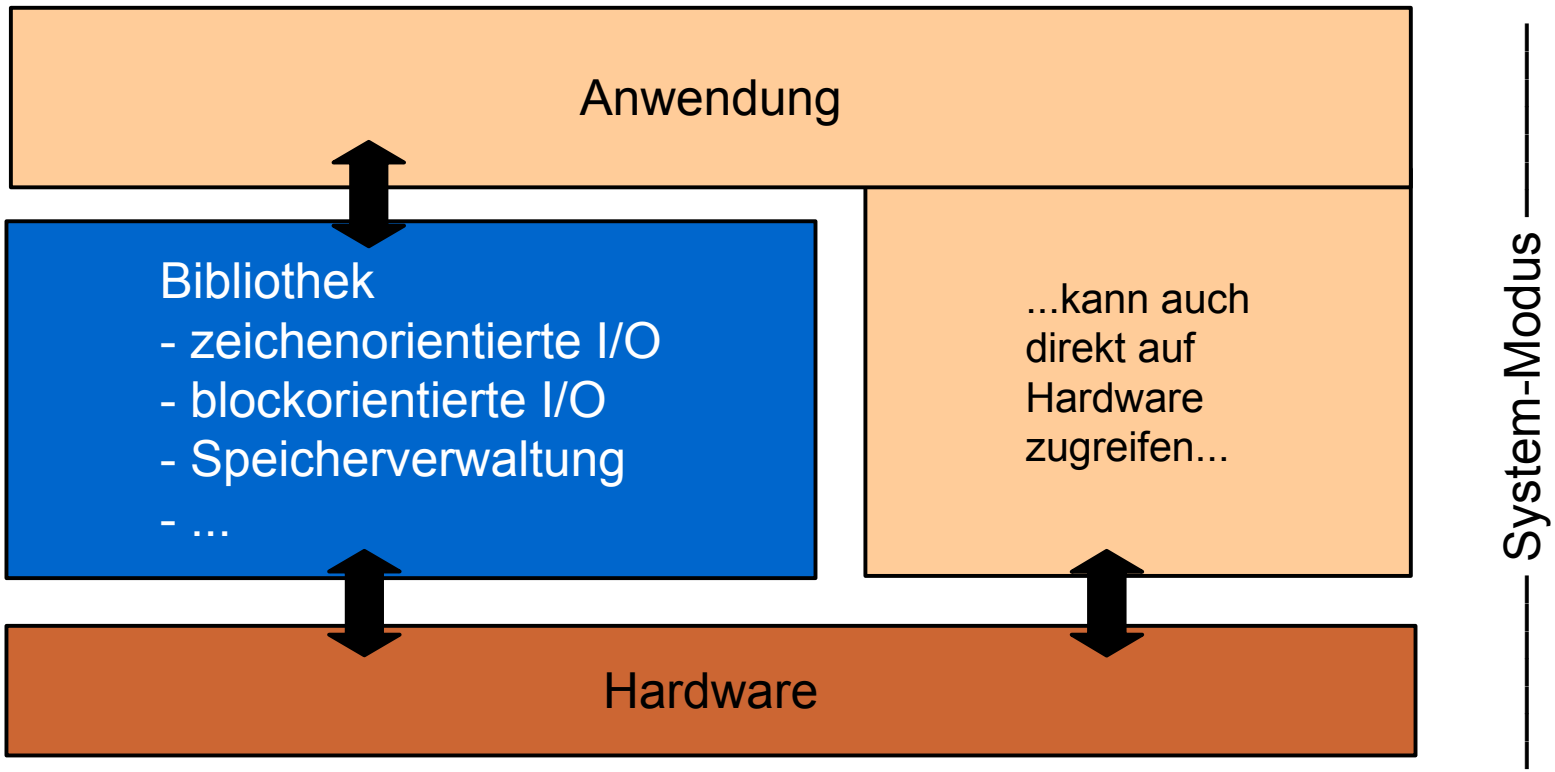
Entstehung von Bibliotheks-Betriebssystemen

- Erste Rechnersysteme besaßen keinerlei Systemsoftware
  - Jedes Programm musste die gesamte Hardware selbst ansteuern
  - Systeme liefen Operator-gesteuert im Stapelbetrieb
    - single tasking, Lochkarten
  - Peripherie war vergleichsweise einfach
    - Seriell angesteuerter Lochkartenleser und -schreiber, Drucker, Bandlaufwerk
- Code zur Geräteansteuerung wurde in jedem Anwendungsprogramm repliziert
  - Die Folge war eine massive Verschwendung von
    - Entwicklungszeit (teuer!)
    - Übersetzungszeit (sehr teuer!)
    - Speicherplatz (teuer!)
  - außerdem eine hohe Fehleranfälligkeit

Entstehung von Bibliotheks-Betriebssystemen

- **Logische Folge:** Bibliotheks-Betriebssysteme
  - Zusammenfassung von häufig benutzen Funktionen zur Ansteuerung von Geräten in Software-Bibliotheken (*Libraries*)
    - Systemfunktionen als „normale” Subroutinen
  - Funktionen der Bibliothek waren dokumentiert und getestet
    - verringerte Entwicklungszeit (von Anwendungen)
    - verringerte Übersetzungszeit (von Anwendungen)
  - Bibliotheken konnten resident im Speicher des Rechners bleiben
    - verringerter Speicherbedarf (der Anwendungen)
    - verringerte Ladezeit (der Anwendungen)
  - Fehler konnten von Experten zentral behoben werden
    - verbesserte Zuverlässigkeit

Architektur: Bibliotheks-Betriebssysteme



Bewertung: Bibliotheks-Betriebssysteme

- Anwendungsorientierte Kriterien
  - **Portabilität** — **gering**
    - keine Standards, eigene Bibliotheken für jede Architektur
  - **Erweiterbarkeit** — **mäßig**
    - theoretisch gut, in der Praxis oft „Spaghetti-Code“
  - **Robustheit** — **sehr hoch**
    - *single tasking*, Kosten für Programmwechsel sehr hoch
  - **Leistung** — **sehr hoch**
    - direktes Operieren auf der Hardware, keine Privilegebenen
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus** — **nicht erforderlich**
    - Anwendung $\equiv$ System
  - **Interaktionsmechanismus** — **Funktionsaufrufe**
    - Betriebssystem $\equiv$ Bibiliothek
  - **Unterbrechungsmechanismus** — **oft nicht vorhanden**
    - Kommunikation mit Geräten über *polling*

Probleme: Bibliotheks-Betriebssysteme

- Teure Hardware wird nicht optimal ausgelastet
  - Hoher Zeitaufwand beim Wechseln der Anwendung
  - Warten auf Ein-/Ausgabe verschwendet unnötig CPU-Zeit
- Organisatorische Abläufe sehr langwierig
  - Stapelbetrieb, Warteschlangen
  - von der Abgabe eines Programms bis zum Erhalt der Ergebnisse vergehen oft Tage – um dann festzustellen, dass das Programm in der ersten Zeile einen Fehler hatte...
- Keine Interaktivität möglich
  - Betrieb durch Operatoren, kein direkter Zugang zur Hardware
  - Programmabläufe nicht zur Laufzeit parametrierbar

Agenda

Einführung

Geschichte, Mode und Trend

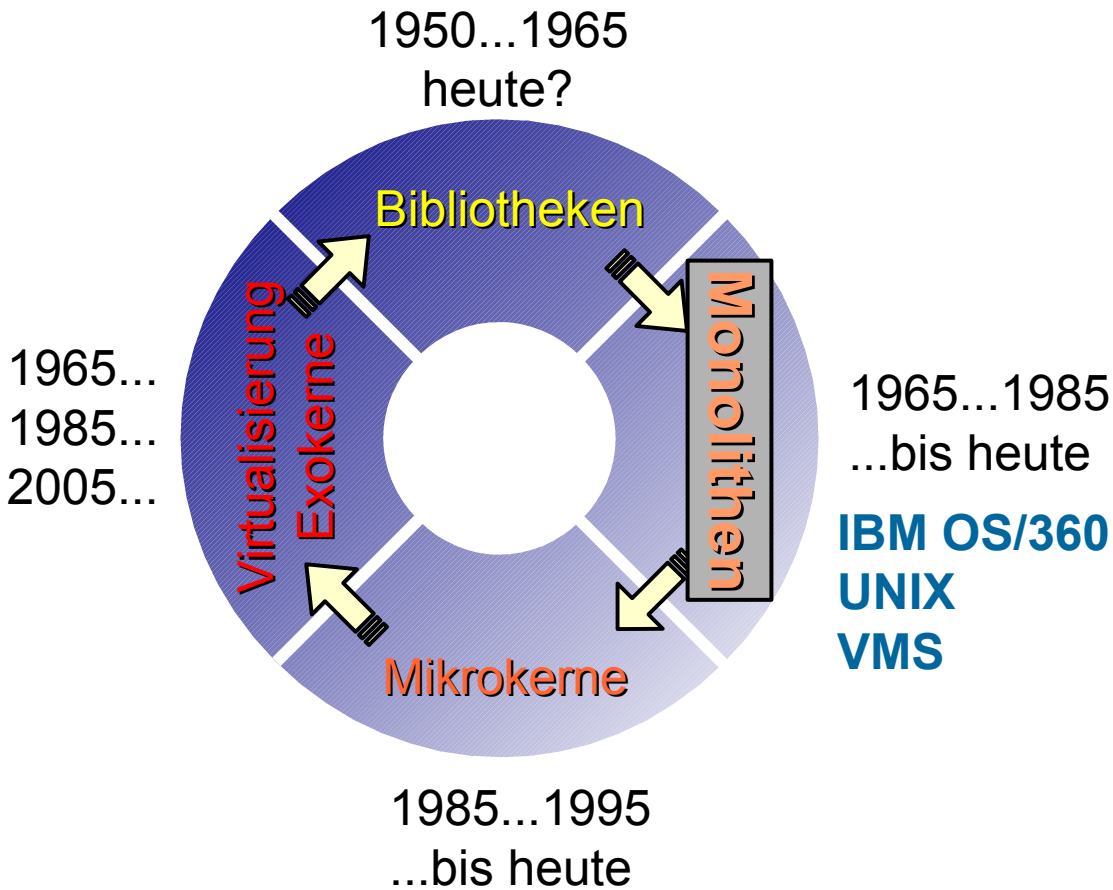
- Bibliotheks-Betriebssysteme
- Monolithen
- Mikrokerne
- Exokerne und Virtualisierung

Zusammenfassung

Referenzen

Überblick: Paradigmen

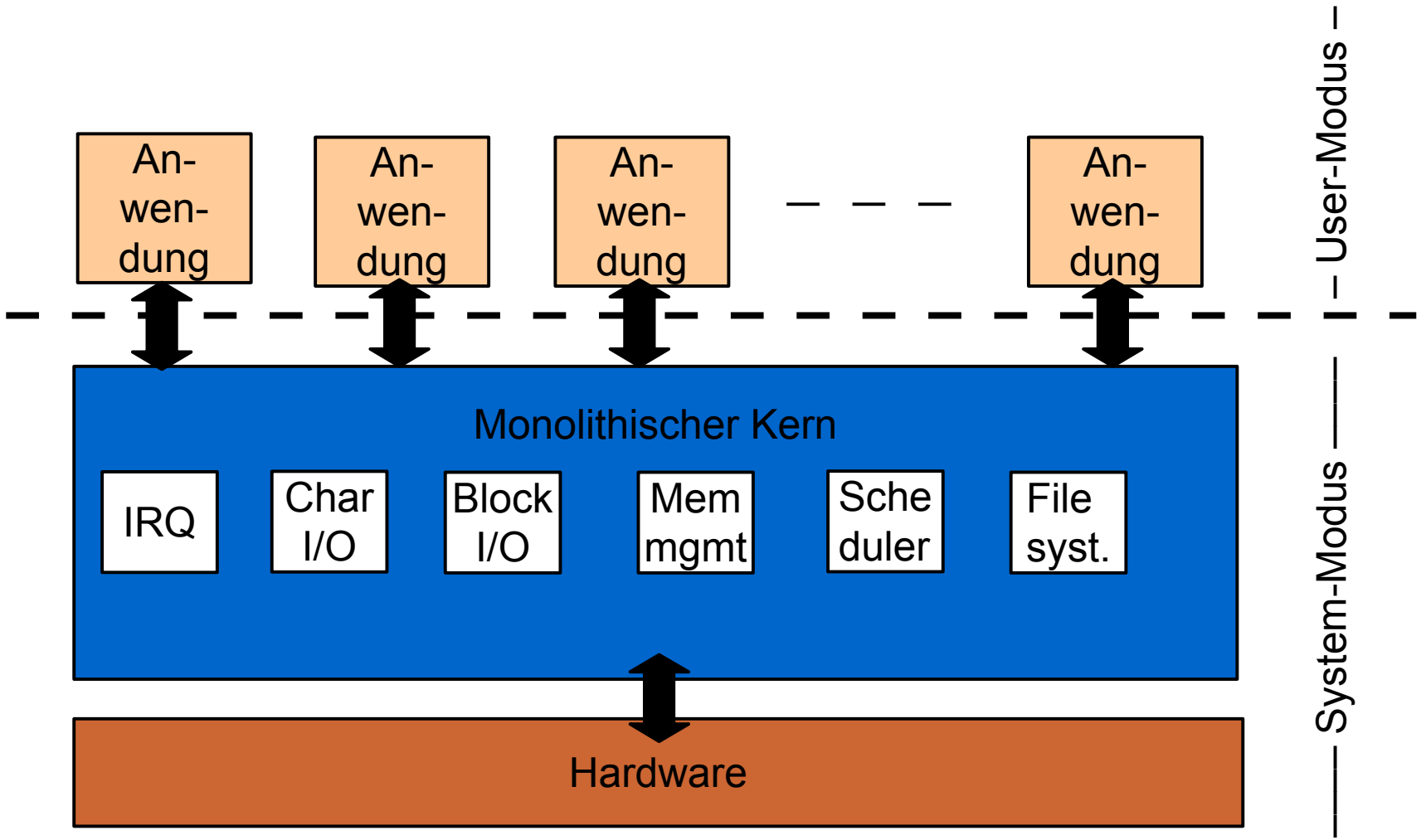
Monolithen als Herrscher über das System



Betriebssystem-Monolithen

- **Motivation:** Mehrprogrammbetrieb
- **Problem:** Isolation
- **Ansatz:** BS als Super-Programm, Kontrollinstanz
  - Programme laufen unter der Kontrolle des Betriebssystems
  - Dadurch erstmals (sinnvoll) Mehrprozess-Systeme realisierbar
- Einführung eines Privilegiensystems
  - Systemmodus $\longleftrightarrow$ Anwendungsmodus
  - Direkter Hardwarezugriff nur im Systemmodus
    $\leadsto$ Gerätetreiber gehören zum System
- Einführung neuer Hard- und Software-Mechansimen
  - *Traps* in den Kern
  - Kontextumschaltung und -sicherung
  - *Scheduling* der Betriebsmittel

Architektur: Monolithische Systeme

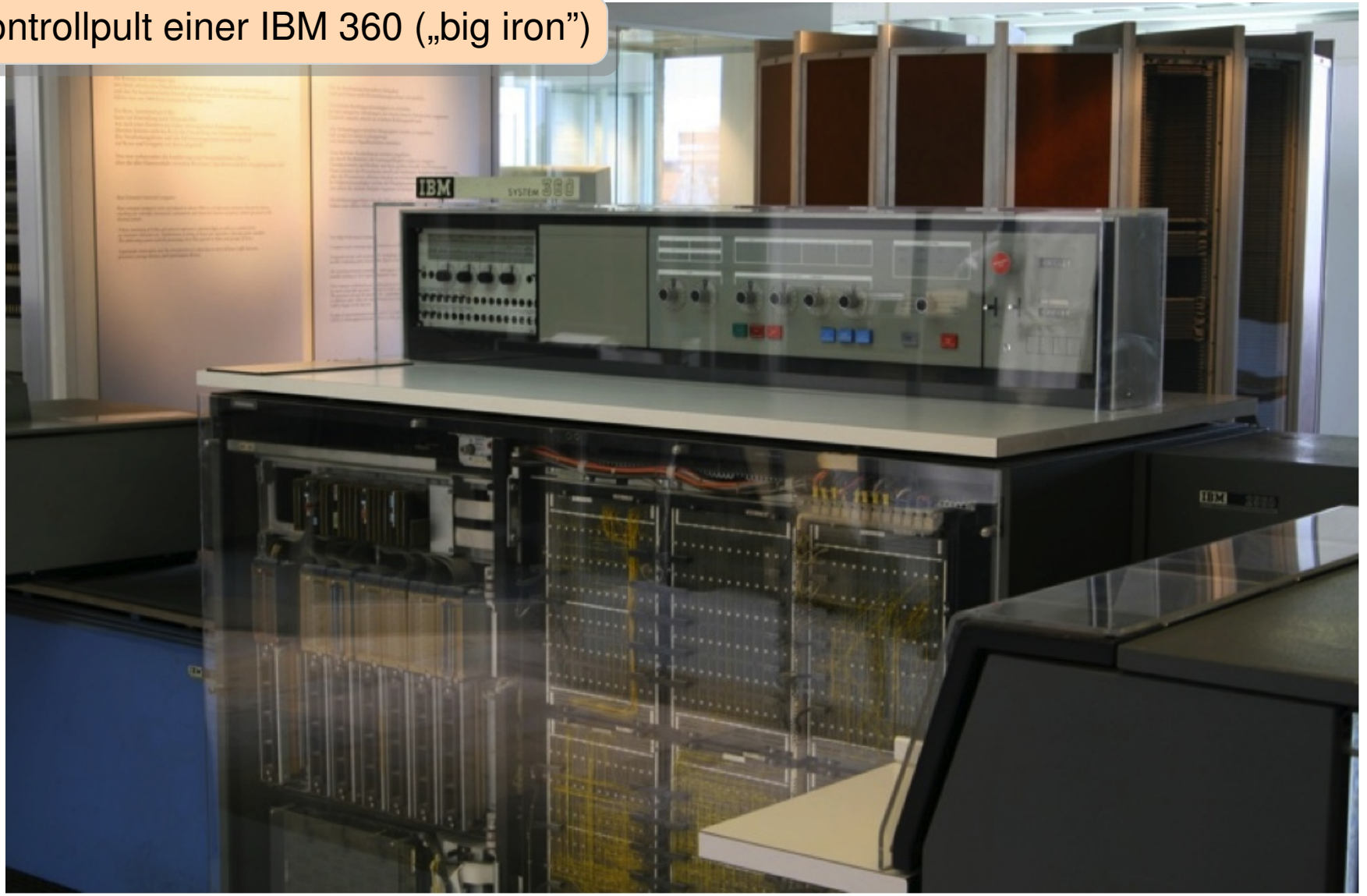


Monolithische Systeme: IBM OS/360

- Eines der ersten monolithischen Betriebssysteme
  - Ziel: gemeinsames BS für alle IBM-Großrechner
  - Leistung und Speicher der Systeme differierten aber um Zehnerpotenzen zwischen „kleinen” und „großen” 360-Systemen
- Diverse Konfigurationen
  - PCP (Primary Control Program) 1965
    - Einprozessbetrieb, kleine Systeme
  - MFT (Multiprogramming with Fixed number of Tasks) 1966
    - mittlere Systeme (256 kB RAM)
    - feste Speicherpartitionierung zwischen Prozessen, feste Anzahl an Tasks
  - MVT (Multiprogramming with Variable number of Tasks): 1967
    - high end
    - Paging, optional Time Sharing Option (TSO) für interaktive Nutzung

Monolithische Systeme: IBM OS/360

Kontrollpult einer IBM 360 („big iron”)



Monolithische Systeme: IBM OS/360

- Richtungsweisende Konzepte
  - Hierarchisches Dateisystem
  - Prozesse können Unterprozesse erzeugen
  - Familienansatz: MFT und MVT sind von API und ABI her kompatibel
- Große Probleme bei der Entwicklung
  - Fred Brooks: „The Mythical Man-Month” [3] lesenswert!
  - Problem der Konzeptuellen Integrität
    - Separation von Architektur und Implementierung war schwierig
  - „Second System Effect”
    - Entwickler wollten die „eierlegende Wollmilchsau” bauen
  - Zu komplexe Abhängigkeiten zwischen Komponenten des Systems
    - Ab einer gewissen Codegröße blieb die Anzahl der Fehler konstant

⇝ Geburt der **Softwaretechnik**

Monolithische Systeme: Bell Labs/AT&T UNIX

- Ziel: Mehrprogrammbetrieb auf „kleinen” Computern
  - Entwicklung seit Anfang der 70er Jahre
    - Kernelgröße im Jahr 1979 (7th Edition Unix, PDP11): ca. 50kB
  - von ursprünglich 2-3 Entwicklern geschrieben
    - überschaubar und handhabbar, ca. 10.000 Zeilen Quelltext
- Neu: Portabilität durch Hochsprache
  - **C** als domänenspezifische Sprache für Systemsoftware
  - UNIX wurde mit den Jahren auf nahezu jede Plattform portiert
- Weitere richtungsweisende Konzepte:
  - alles ist eine Datei, dargestellt als ein Strom von Bytes
  - komplexe Prozesse werden aus einfachen Programmen komponiert
    - Konzept der Pipe, Datenflussparadigma

Monolithische Systeme: Bell Labs/AT&T UNIX

Historische PDP11

Monolithische Systeme: Bell Labs/AT&T UNIX

- Weitere Entwicklung von UNIX erfolgte stürmisch
  - Systeme mit großem Adressraum (VAX, RISC)
  - Der Kernel ist „mit gewachsen” (System III, System V, BSD)
    - ohne wesentliche Strukturänderungen
  - Immer mehr komplexe Subsysteme wurden integriert
    - TCP/IP ist ungefähr so umfangreich wie der Rest des Kernels
- Linux orientiert(e) sich an der Struktur von System V
- UNIX war und ist einflussreich im akademischen Bereich durch frühe „Open Source”-Politik der Bell Labs
  - Viele Portierungen und Varianten entstanden
    - oftmals parallel zu Hardwareentwicklungen
  - In der akademischen Welt wurde UNIX zum Referenzsystem
    - Ausgleichspunkt und Vergleichssystem für alle neueren Ansätze

Bewertung: Betriebssystem-Monolithen

- Anwendungsorientierte Kriterien
  - **Portabilität** **hoch**
    - dank „C” kann und konnte UNIX einfach portiert werden
  - **Erweiterbarkeit** **mäßig**
    - von Neukompilierung $\rightsquigarrow$ Modulkonzept
  - **Robustheit** **mäßig**
    - Anwendungen isoliert, nicht jedoch BS-Komponenten (Treiber!)
  - **Leistung** **hoch**
    - Nur Betreten / Verlassen des Kerns ist teuer
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus** **Privilegebenen, Adressräume**
    - Pro Anwendung ein Adressraum, Kern läuft auf Systemebene
  - **Interaktionsmechanismus** **Funktionsaufrufe, Traps**
    - Anwendung $\rightarrow$ Kern durch *Traps*, innerhalb des Kerns durch `call` / `ret`
  - **Unterbrechungsmechanismus** **Bearbeitung im Kern**
    - interne Unterteilung in UNIX: *bottom half*, *top half*

Probleme: Betriebssystem-Monolithen

- Monolithen sind schwer handhabbar
  - Hinzufügen oder Abändern von Funktionalität betrifft oft mehr Module, als der Entwickler vorhergesehen hat
- Eingeschränkte Synchronisationsmechanismen
  - Oft nur ein „Big Kernel Lock”, d. h. nur ein Prozess kann zur selben Zeit im Kernmodus ausgeführt werden, alle anderen warten
  - Insbesondere bei Mehrprozessor-Systemen leistungsreduzierend
- Gemeinsamer Adressraum aller Kernkomponenten
  - Sicherheitsprobleme in einer Komponente (z.B. buffer overflow) führen zur Kompromittierung des gesamten Systems
  - Viele Komponenten laufen überflüssigerweise im Systemmodus
  - Komplexität und Anzahl von Treibern hat extrem zugenommen

Agenda

Einführung

Geschichte, Mode und Trend

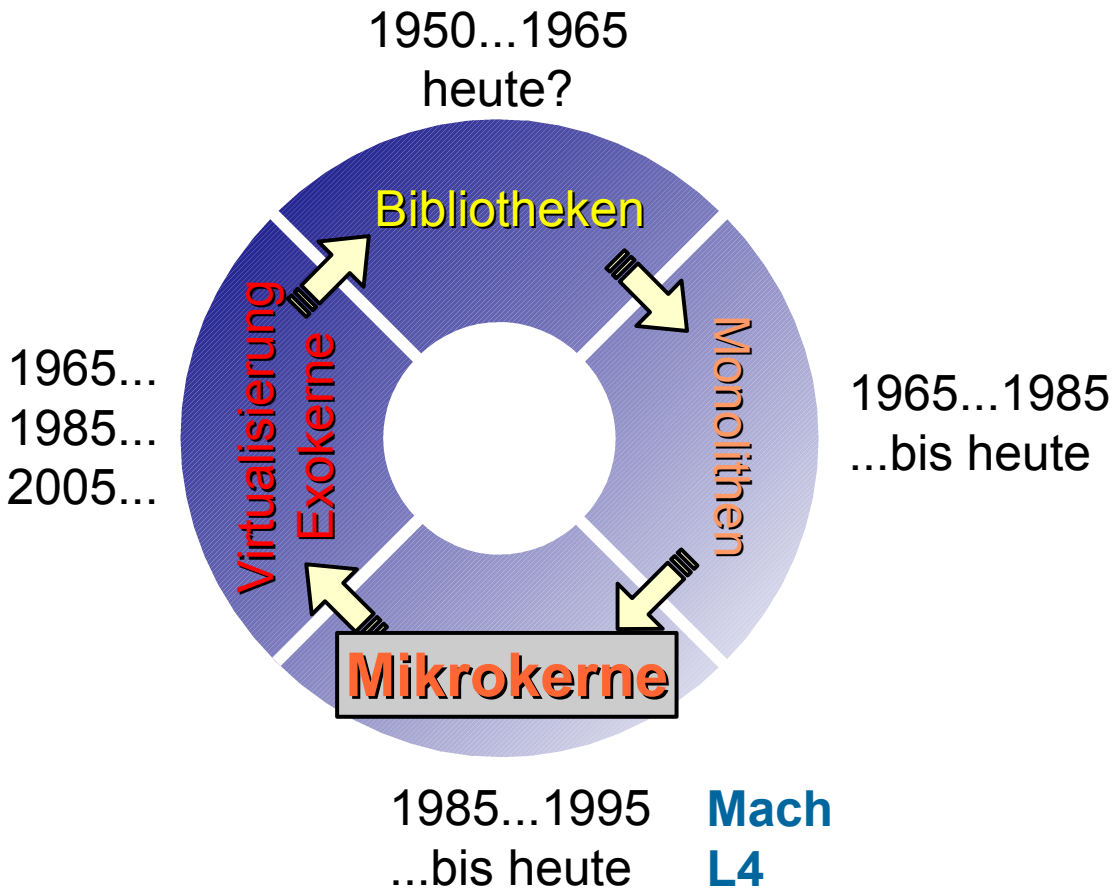
- Bibliotheks-Betriebssysteme
- Monolithen
- Mikrokerne
- Exokerne und Virtualisierung

Zusammenfassung

Referenzen

Überblick: Paradigmen

Mikrokerne als Reduktion auf das Notwendige



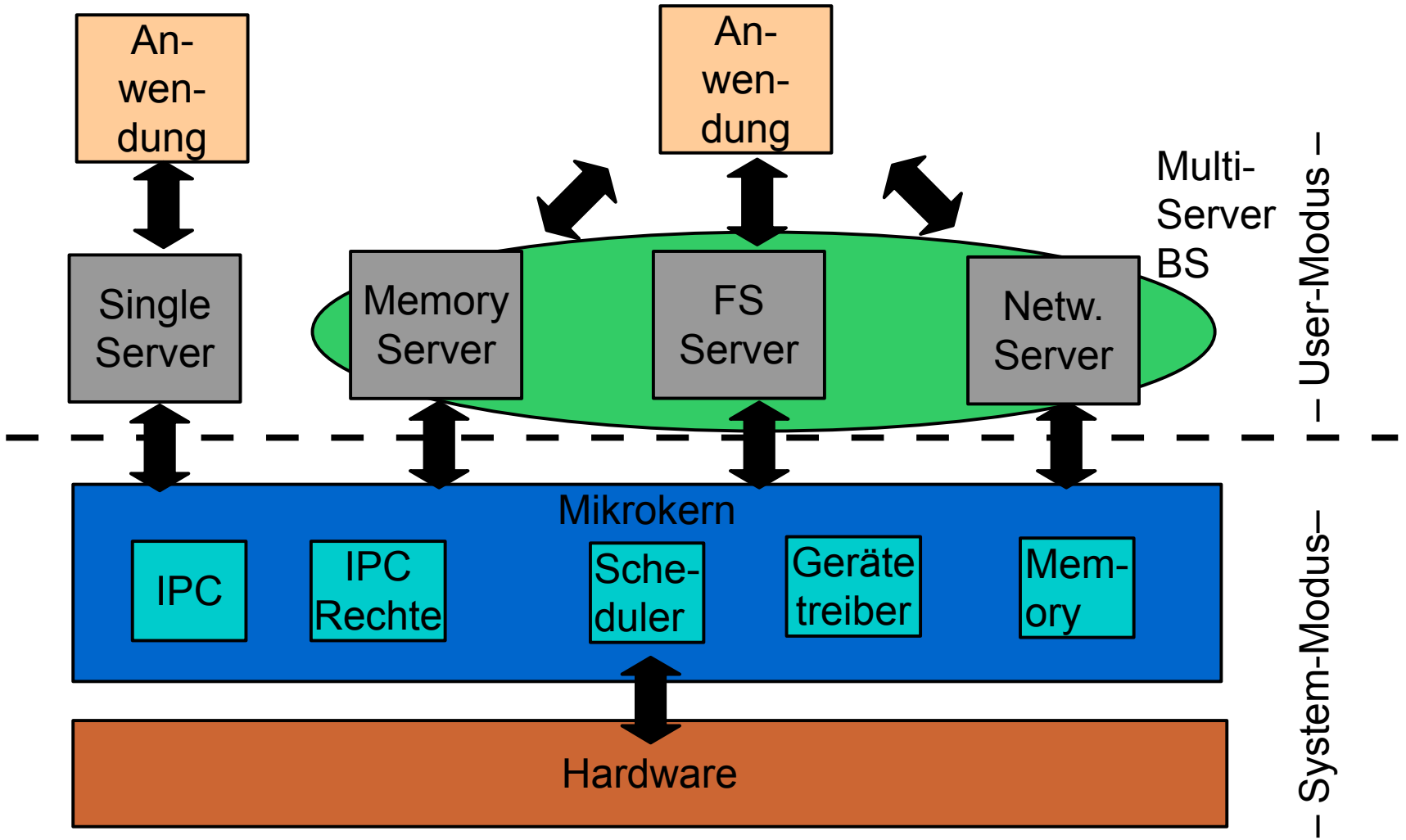
Mikrokern-Betriebssysteme

- Ziel: Reduktion der Trusted Computing Base (TCB)
  - Minimierung der im privilegierten Modus ablaufenden Funktionalität
  - BS-Komponenten als Server-Prozesse im nichtprivilegierten Modus
  - Interaktion über Nachrichten (IPC, *Inter Process Communication*)
- Prinzip des geringsten Privilegs
  - Systemkomponenten müssen nur so viele Privilegien besitzen, wie zur Ausführung ihrer Aufgabe erforderlich sind
    - z.B. Treiber: Zugriff auf spezielle IO-Register, nicht auf die gesamte HW
  - Nur der Mikrokern läuft im Systemmodus
- Geringere Codegröße
  - L4: 10 kloc C++ $\longleftrightarrow$ Linux: 1 Mloc C (ohne Treiber)
  - Ermöglicht Ansätze zur formalen Verifikation des Mikrokerns [6]

Mikrokerne erster Generation: CMU Mach [1]

- **Ziel:** Reduktion der TCB
- **Ziel:** Schaffung eines extrem portablen Systems
- **Ziel:** Verbesserung der Unix-Konzepte
  - Neue Kommunikationsmechanismen via IPC und Ports
    - Ports sind sichere IPC-Kommunikationskanäle
    - IPC ist optional netzwerktransparent: Unterstützung für verteilte Systeme
  - Parallele Aktivitäten innerhalb eines Prozessadressraums
    - Unterstützung für Fäden $\rightsquigarrow$ neuer Prozessbegriff als „Container”
    - Bessere Unterstützung für Mehrprozessorsysteme
  - Unterstützung „fremder” Systemschnittstellen durch Personalities
- Ausgangspunkt: BSD UNIX
  - Schrittweise Separation der Funktionalität, die nicht im privilegierten Modus laufen muss in Benutzermodus-Prozesse
  - Anbindung über Ports und IPC

Architektur: Mikrokerne erster Generation



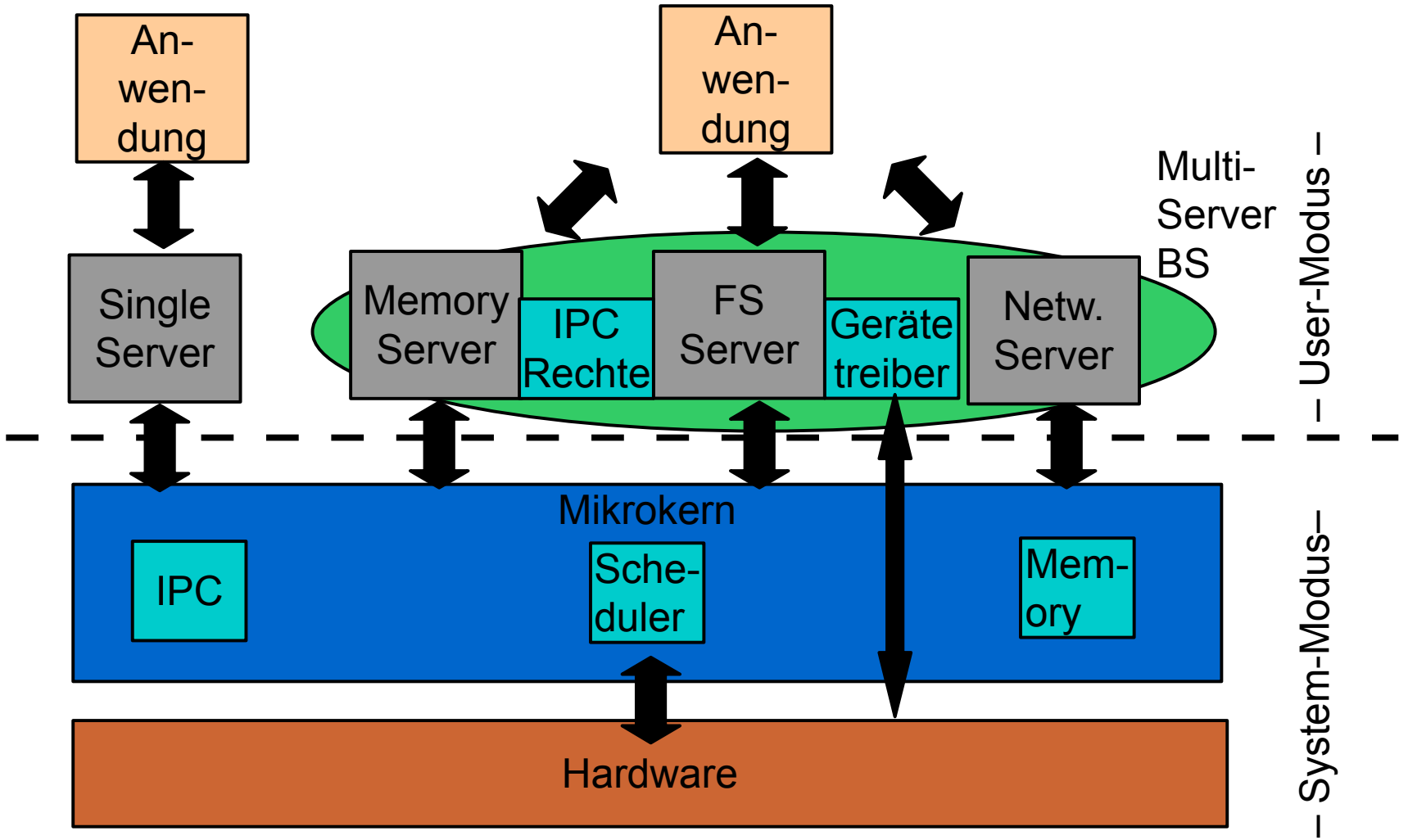
Probleme: Mikrokerne erster Generation

- Probleme von Mach
  - hoher Overhead für IPC-Operationen
    - Systemaufrufe Faktor 10 langsamer gegenüber monolithischem Kern
  - Immer noch viel zu große Code-Basis
    - Gerätetreiber und Rechteverwaltung für IPC im Mikrokern
    - $\leadsto$ die eigentlichen Probleme nicht gelöst!
  - Führte zu schlechtem Ruf von Mikrokernen allgemein
    - Einsetzbarkeit in der Praxis wurde bezweifelt
- Die Mikrokern-Idee galt Mitte der 90er Jahre als tot
  - Praktischer Einsatz von Mach erfolgte nur in hybriden Systemen
    - Separat entwickelte Komponenten für Mikrokern und Server
    - Kolokation der Komponenten in einem Adressraum,
      Ersetzen von in-kernel IPC durch Funktionsaufrufe
  - Bekanntestes Beispiel: Apple OS X $\mapsto$ Mach 3 Mikrokern + FreeBSD

Mikrokerne zweiter Generation: L4 [5, 7]

- **Ziel:** Mikrokern, diesmal aber richtig!
  - Verzicht auf Sekundärziele: Portabilität, Netzwerktransparenz, ...
- **Ansatz:** Reduktion auf das Notwendigste
  - Ein Konzept wird nur dann innerhalb des Mikrokerns toleriert, wenn seine Auslagerung die Implementierung verhindern würde.
  - synchroner IPC, Kontextwechsel, CPU Scheduler, Adressräume
- **Ansatz:** Gezielte Beschleunigung
  - fast IPCs durch Parameterübergabe in Registern
  - Gezielte Reduktion der Cache-Load (durch sehr kleinen Kern)
- Viele von Mikrokernen der 1. Generation noch im Systemmodus implementierte Funktionalität ausgelagert
  - z. B. Überprüfung von IPC-Kommunikationsrechten
  - vor allem aber: Treiber

Architektur: Mikrokerne zweiter Generation



Bewertung: Mikrokern-Betriebssysteme

- Anwendungsorientierte Kriterien
  - **Portabilität** — **mäßig**
    - ursprünglich rein in Assembler, aktuell in C++ entwickelt
  - **Erweiterbarkeit** — **sehr hoch**
    - durch neue Server im Benutzermodus, auch zur Laufzeit
  - **Robustheit** — **sehr hoch**
    - durch strikte Isolierung
  - **Leistung** — **mäßig – gut**
    - IPC-Performance ist **der** kritische Faktor
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus** — **Adressräume**
    - Ein Adressraum pro Anwendung,
      ein Adressraum pro Systemkomponente
  - **Interaktionsmechanismus** — **IPC**
    - Anwendungen und Systemkomponenten interagieren über Nachrichten
  - **Unterbrechungsmechanismus** — **IPC an Server-Prozess**
    - Unterbrechungsbehandlung erfolgt durch Faden im Benutzermodus

Agenda

Einführung

Geschichte, Mode und Trend

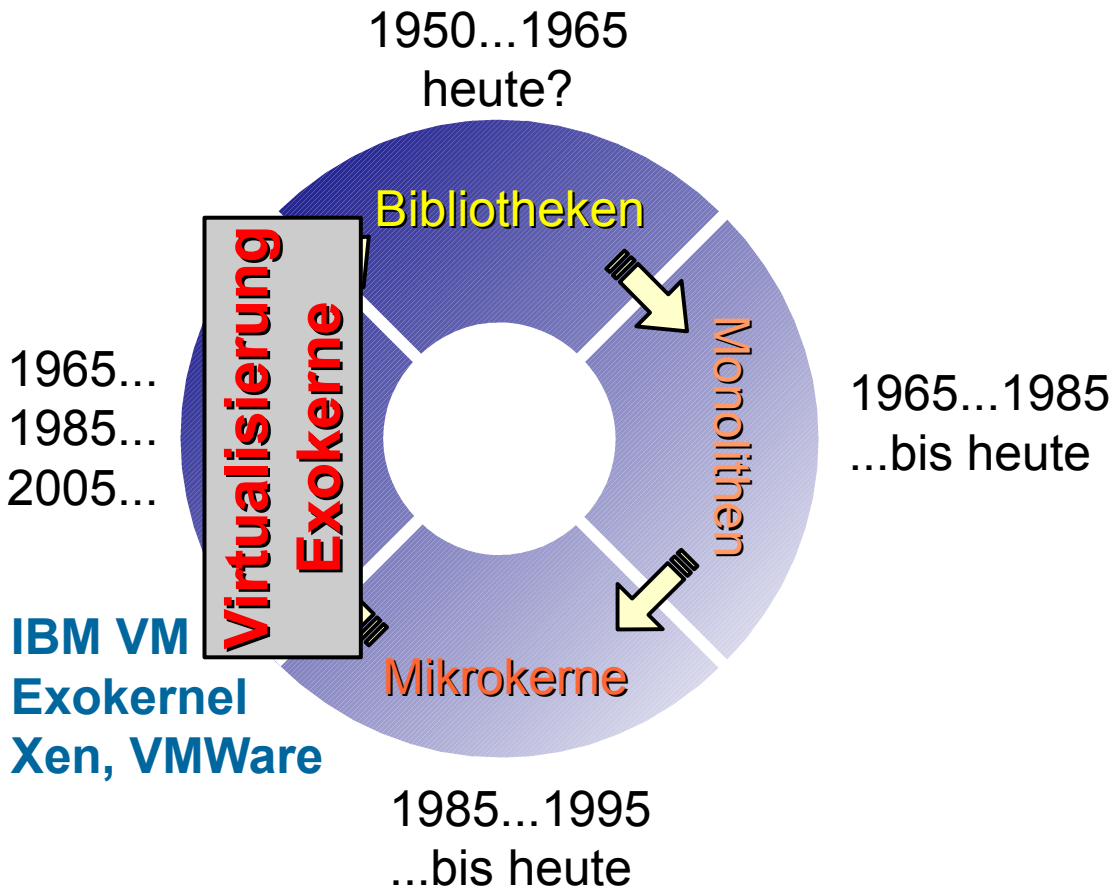
- Bibliotheks-Betriebssysteme
- Monolithen
- Mikrokerne
- Exokerne und Virtualisierung

Zusammenfassung

Referenzen

Überblick: Paradigmen

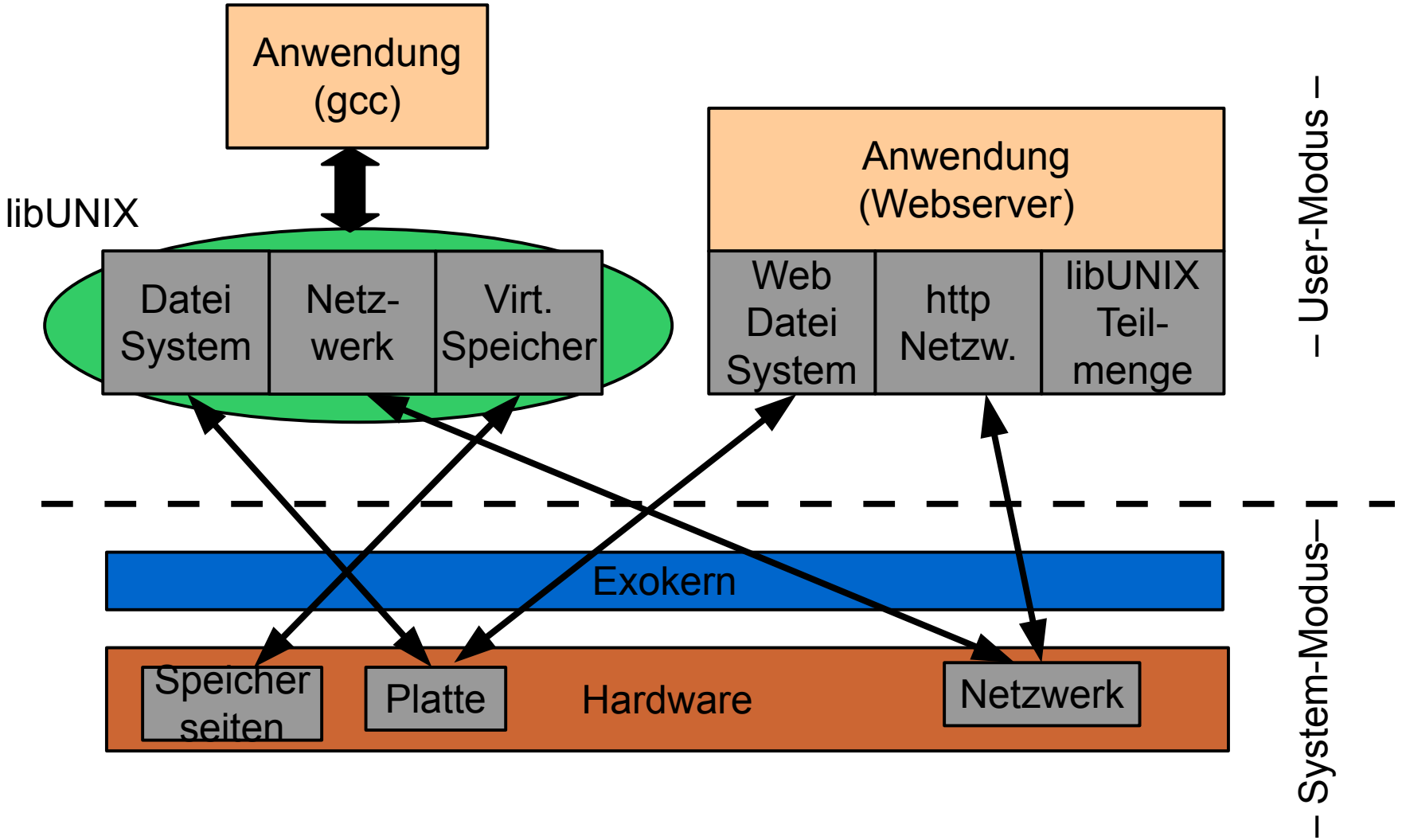
Exokerne und Virtualisierung als weitere Reduktion



Exokern-Betriebssysteme: MIT exokernel [4]

- Ziel: Leistungsverbesserung durch Reduktion
  - Entfernung von Abstraktionsebenen
  - Implementierung von Strategien (z.B. Scheduling) in der Anwendung
- Extrem kleiner Kern, dieser Implementiert nur
  - Schutz
  - Multiplexing von Ressourcen (CPU, Speicher, Disk-Blöcke, ...)
- Trennung von Schutz und Verwaltung der Ressourcen!
  - Keine Implementierung von IPC-Mechanismen (Mikrokerne) oder weiterer Abstraktionen (Monolithen)
  - Anwendungen können die *für sie* idealen Abstraktionen, Komponenten und Strategien verwenden

Architektur: Exokern-Betriebssysteme



Bewertung: Exokern-Betriebssysteme

- Anwendungsorientierte Kriterien
  - **Portabilität** — **sehr hoch**
    - Exokerne sind sehr klein
  - **Erweiterbarkeit** — **sehr hoch**
    - aber auch erforderlich! – der Exokern stellt kaum Funktionalität bereit
  - **Robustheit** — **gut**
    - Schutz wird durch den Exokern bereitsgestellt
  - **Leistung** — **sehr gut**
    - Anwendungen operieren nahe an der Hardware, wenige Abstraktionsebenen
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus** — **Adressräume**
    - Ein Adressraum pro Anwendung
      + von ihr gebrauchter Systemkomponenten
  - **Interaktionsmechanismus** — **nicht vorgegeben**
    - wird von der Anwendung bestimmt
  - **Unterbrechungsmechanismus** — **nicht vorgegeben**
    - Exokern verhindert nur die Monopolisierung der CPU

Probleme: Exokern-Betriebssysteme

- Exokernel sind nicht als Basis für die Verwendung mit beliebigen „legacy”-Anwendungen geeignet
- Anwendungen haben volle Kontrolle über Abstraktionen
  - müssen diese aber auch implementieren
  - hohe Anforderungen an Anwendungsentwickler
- Definition von Exokern-Schnittstellen ist schwierig
  - Bereitstellung adäquater Schnittstellen zur System-Hardware
  - Genaue Abwägung zwischen Mächtigkeit, Minimalismus und ausreichendem Schutz
- Bisher kein Einsatz in Produktionssystemen
  - Es existieren lediglich einige *proof-of-concept-Systeme*
  - Viele Fragen der Entwicklung von BS-Bibliotheken noch offen

Virtualisierung [10]

- Ziel: Isolation und Multiplexing
  unterhalb der Systemebene
- Ansatz: Virtual Machine Monitor (VMM) / Hypervisor
  - Softwarekomponente, läuft direkt auf der Hardware
  - stellt Abstraktion Virtual Maschine (VM) zur Verfügung
- VM simuliert die gesamten Hardware-Ressourcen
  - Prozessoren, Speicher, Festplatten, Netzwerkkarten, ...
  - Container für beliebige Betriebssysteme nebst Anwendungen
- Vergleich zu Exokernen
  - gröbere Granularität der zugeteilten Ressourcen
    - z.B. gesamte Festplattenpartition vs. einzelne Blöcke
  - „brute force” Ansatz
    - Multiplexen ganzer Rechner statt einzelner Betriebsmittel
  - Anwendungen (und BS) brauchen nicht angepasst werden

Virtualisierung: Beispiel IBM VM/370 (1972)

- Für IBM 360-Großrechner existierten mehrere Betriebssysteme
  - DOS/360, MVS: Stapel-orientierte Bibliotheks-Betriebssysteme
  - OS/360: Midrange Server-System
  - TSS/360: Interaktives Mehrbenutzersystem mit Time-Sharing
  - Kundenspezifische Entwicklungen
- Problem: wie kann man Anwendungen für all diese Systeme *gleichzeitig* verwenden?
  - Hardware war teuer (Millionen von USD)
- Entwicklung der ersten Systemvirtualisierung „VM” durch Kombination aus Emulation und Hardware-Unterstützung
  - Harte Partionierung der Betriebsmittel
  - Gleichzeitiger Betrieb von stapelverarbeitenden und interaktiven Betriebssystemen wurde ermöglicht

Virtualisierung von PCs: Beispiele VMWare, Xen (2003) [2]

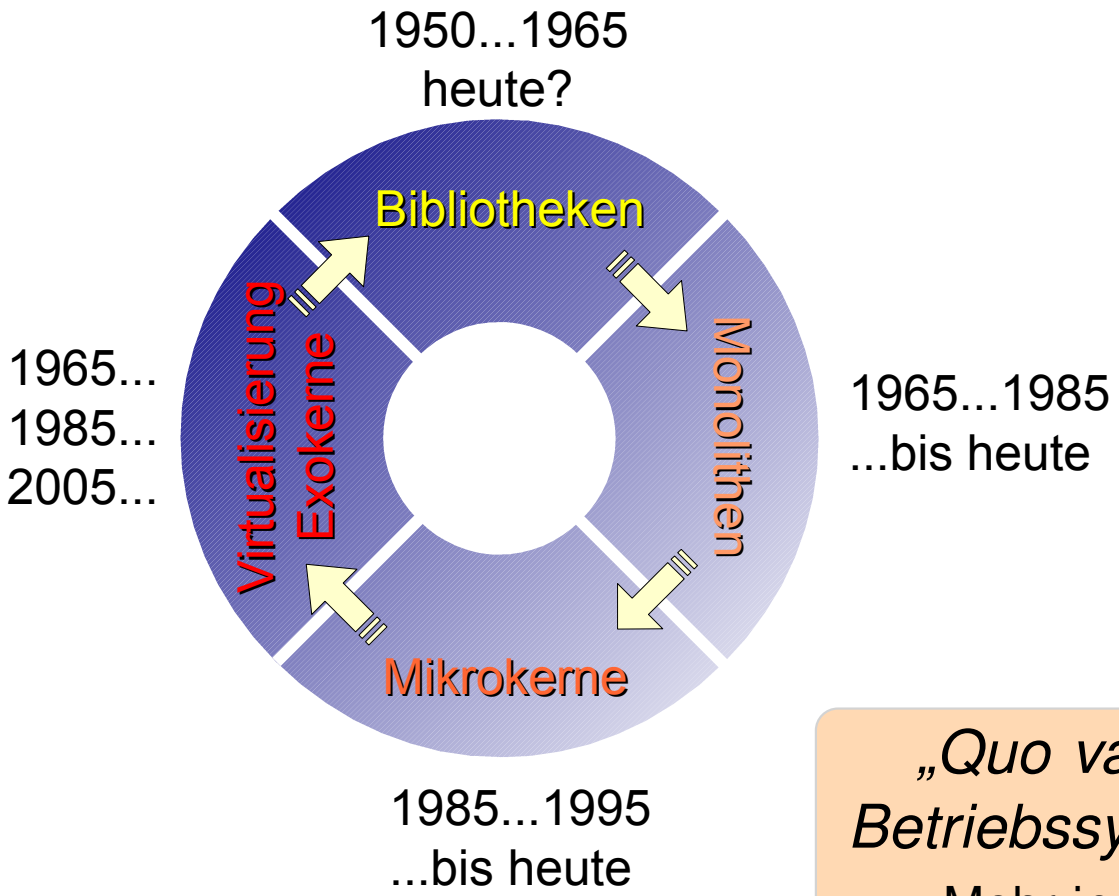
- Ausgangslange: Problematik wie bei IBM in den 60er Jahren
  - Hardware wird immer leistungsfähiger – wohin mit den Ressourcen?
  - Ablauf mehrerer Betriebssystem-Instanzen gleichzeitig
  - Serverkonsolidierung, Kompatibilität zu Anwendungen
- Problem: IA-32 ist eigentlich nicht virtualisierbar
  - Virtualisierungskriterien von Popek und Goldberg [8] sind nicht erfüllt
  - Insbesondere: Äquivalenzanforderung – nicht alle Ring 0 Befehle trappen bei Ausführung auf Ring 3
- Ansatz: Paravirtualisierung
  - „kritische Befehle” werden ersetzt
    - entweder zur Übersetzungszeit (Xen) oder zur Laufzeit (VMWare)
  - VMs laufen in Ring 3, Ringmodell durch Adressräume nachgebildet
    - Die meisten BS verwenden eh nur Ring 0 und Ring 3
- Neue IA-32 CPUs unterstützen Virtualisierung in HW (⇝ VL 6)
  - Paravirtualisierung in der Praxis oft noch perfomanter

Bewertung: Virtualisierung

- Anwendungsorientierte Kriterien
  - **Portabilität** — **gering**
    - sehr hardware-spezifisch, Paravirtualisierung ist aufwändig
  - **Erweiterbarkeit** — **keine**
    - in den üblichen VMMs nicht vorgesehen
  - **Robustheit** — **gut**
    - grobgranular auf der Ebene von VMs
  - **Leistung** — **mäßig – gut**
    - stark abhängig vom Einsatzszenario (CPU-lastig, IO-lastig, . . .)
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus** — **VM, Paravirtualisierung**
    - Jede Instanz bekommt einen eigenen Satz an Hardwaregeräten
  - **Interaktionsmechanismus** — **nicht vorgesehen**
    - Anwendungen in den VMS kommunizieren miteinander über TCP/IP
  - **Unterbrechungsmechanismus** — **Weiterleitung an VM**
    - VMM simuliert Unterbrechungen in den einzelnen VMs

Überblick: Paradigmen

Back where we started?



„Quo vadis", Betriebssysteme

⇝ Mehr in VL 13!

Agenda

Einführung
Geschichte, Mode und Trend
Zusammenfassung
Referenzen

Zusammenfassung: Betriebssystem-Architektur

- Betriebssysteme sind ein unendliches Forschungsthema
  - „alte” Technologien (wie Virtualisierung oder Bibliotheken) finden immer wieder neue Einsatzgebiete
  - Hardwaretechnologie treibt die weitere Entwicklung
- Revolutionäre Neuerungen sind schwer durchzusetzen
  - Kompatibilität ist ein hohes Gut
    - Auf Anwendungsebene durch *Personalities* erreichbar
    - Neue Systeme scheitern jedoch meistens an fehlenden Treibern
  - Virtualisierte Hardware als Kompatibilitätsebene
- Die „ideale” Architektur ist letztlich eine Frage der Anwendung!
  - Sensornetze, tief eingebettete Systeme
    Desktoprechner, Server, . . .
  - Architektur $\mapsto$ nichtfunktionale Eigenschaft des Betriebssystems

Referenzen

Mike Accetta, Robert Baron, David Golub u. a. „Mach: A New Kernel Foundation for UNIX Development“. In: *Proceedings of the USENIX Summer Conference*. Atlanta, GA, USA: USENIX Association, Juni 1986, S. 93–113.

Paul Barham, Boris Dragovic, Keir Fraser u. a. „Xen and the Art of Virtualization“. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Bd. 37, 5. ACM SIGOPS Operating Systems Review. Bolton Landing, NY, USA: ACM Press, Okt. 2003, S. 164–177. DOI: 10.1145/945445.945462.

Fred Brooks. *The Mythical Man Month*. Addison-Wesley, 1975. ISBN: 0-201-00650-2.

Dawson R. Engler, M. Frans Kaashoek und James O'Toole. „Exokernel: An Operating System Architecture for Application-Level Resource Management“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM Press. Copper Mountain, CO, USA: ACM Press, Dez. 1995, S. 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224057.224076.

Hermann Härtig, Michael Hohmuth, Jochen Liedtke u. a. „The Performance of μ-Kernel-Based Systems“. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM Press. Saint Malo, France: ACM Press, Okt. 1997. ISBN: 0-89791-916-5. DOI: 10.1145/269005.266660.

Referenzen (Forts.)

Gerwin Klein, Kevin Elphinstone, Gernot Heiser u. a. „seL4: formal verification of an OS kernel“. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. ACM Press. Big Sky, MT, USA: ACM Press, Okt. 2009, S. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

Jochen Liedtke. „On μ-Kernel Construction“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM Press. Copper Mountain, CO, USA: ACM Press, Dez. 1995, S. 237–250. ISBN: 0-89791-715-4. DOI: 10.1145/224057.224075.

Gerald J. Popek und Robert P. Goldberg. „Formal Requirements for Virtualizable Third Generation Architectures“. In: *Communications of the ACM* 17.7 (1974), S. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073.

Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95). ACM Press. Copper Mountain, CO, USA: ACM Press, Dez. 1995. ISBN: 0-89791-715-4.

Jim Smith und Ravi Nair. *Virtual Machines. Versatile Platforms for Systems and Processes*. Elsevier, 2005. ISBN: 978-1558609105.

Betriebssysteme (BS)

VL 11 – Fadensynchronisation

Volkmar Sieh / **Daniel Lohmann**

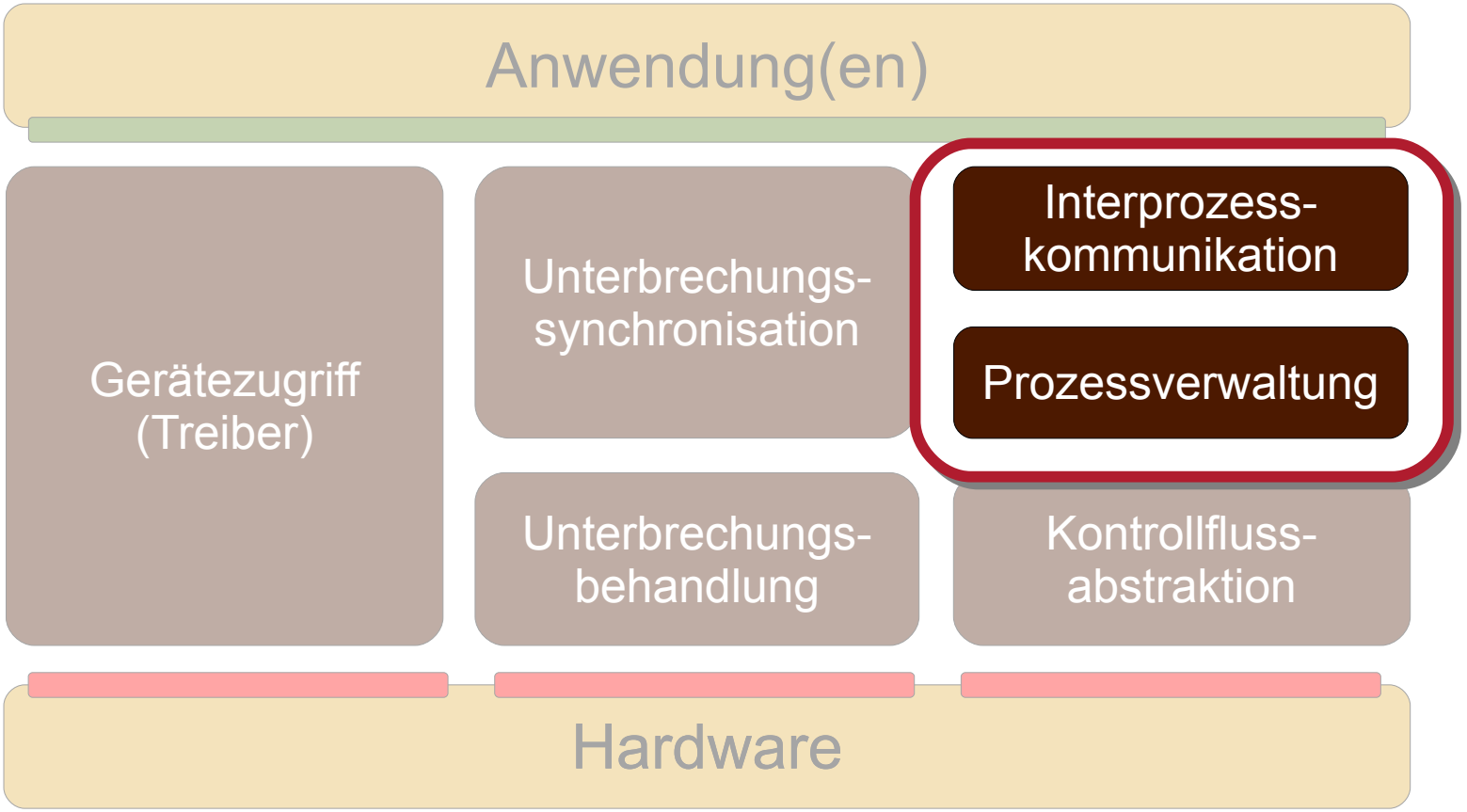
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 17. Januar 2024

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



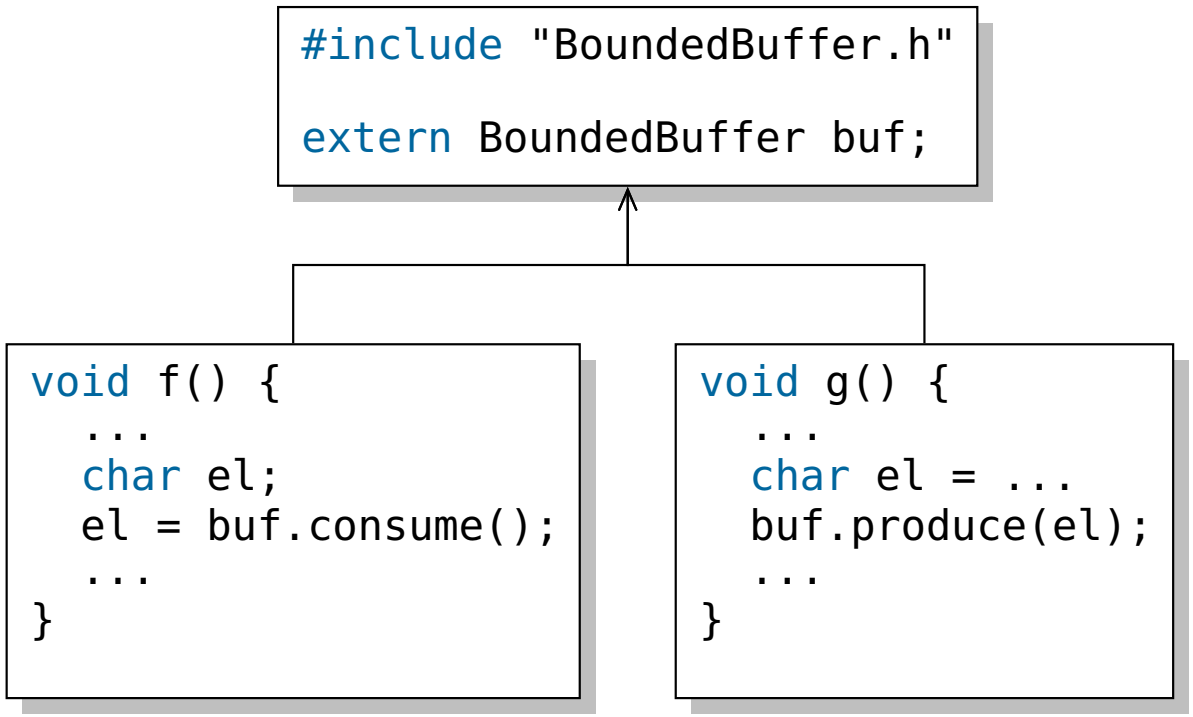
Betriebssystementwicklung

Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
  - Randbedingungen
  - Mutex, Implementierungsvarianten
  - Passives Warten
  - Semaphore
- Beispiel: Windows
  - Warteobjekte
  - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen

Motivation Szenario

- Gegeben: Programmfäden <f> und <g>
  - Präemtives Scheduling (z. B. *round robin*)
  - Zugriff auf gemeinsame Datenstruktur `buf`



Motivation Szenario

- Gegeben: Programmfäden <f> und <g>
  - **Problem:** Pufferzugriffe können überlappen

```
char BoundedBuffer::consume() {
  int elements = occupied;
  if (elements == 0) return 0;
  char result = buf[nextout];
  nextout++; nextout %= SIZE;
```

↯ resume <g>

```
...
void BoundedBuffer::produce(char data) {
  int elements = occupied;
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1;
}
...
```

↯ resume <f>

```
  occupied = elements - 1;
  return result;
}
```

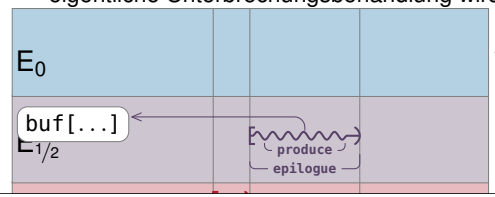
Das hatten wir doch schon mal...

Rückblick: VL 6 – Unterbrechungssynchronisation

Was ist diesmal anders?

Prolog/Epilog-Modell – Ansatz

- **Ansatz:** Latenzverbergung durch zusätzliche Ebene
  - Wir fügen eine weitere *logische* Ebene ein: $E_{1/2}$
    - $E_{1/2}$ liegt zwischen der Anwendungsebene E_0 und den UB-Ebenen $E_{1\ldots n}$
  - Unterbrechungsbehandlung wird zweigeteilt in Prolog und Epilog
    - Prolog arbeitet auf Unterbrechungsebene $E_{1\ldots n}$
    - Epilog arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (Epilogebene)
  - Zustand liegt (so weit wie möglich) auf der Epilogebene
    - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt

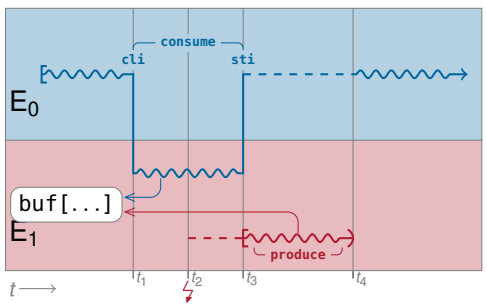


Bounded Buffer – Lösung mit harter Synchronisation

Zugriff „von oben“ wird hart synchronisiert: Für die Ausführung von `consume()` wechselt der Kontrollfluss auf E_1

```
char consume() {
  cli();
  ...
  char result = buf[nextout++];
  ...
  sti();
  return result;
}
```

```
void produce(char data) {
  // hier nichts zu tun
  ...
  buf[nextin++] = data;
  ...
  //hier nichts zu tun
}
```



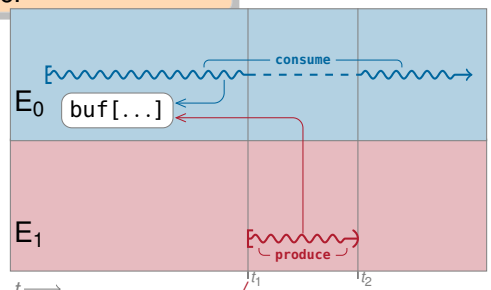
Zustand liegt (logisch) auf E_1

Bounded Buffer – Ansatz mit weicher Synchronisation

Zugriff „von unten“ wird weich synchronisiert: `consume()` liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung `produce()` ausgeführt wurde.

```
char consume() {
  ?
}
```

```
void produce(char data) {
  ?
}
```



Zustand liegt (logisch) auf E_0

Erstes Fazit

- **Bisher:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus verschiedenen Ebenen
  - Zustand wurde auf einer Ebene „platziert”
  - Sicherung entweder „von oben” (hart) oder „von unten” (weich)
  - Innerhalb einer Ebene wurde implizit sequentialisiert
- **Nun:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus derselben Ebene
  - Fäden können jederzeit durch andere Fäden verdrängt werden
  - Fäden können echt parallel arbeiteten (bei mehreren CPUs)

Das ist ja auch der Sinn von Fäden!

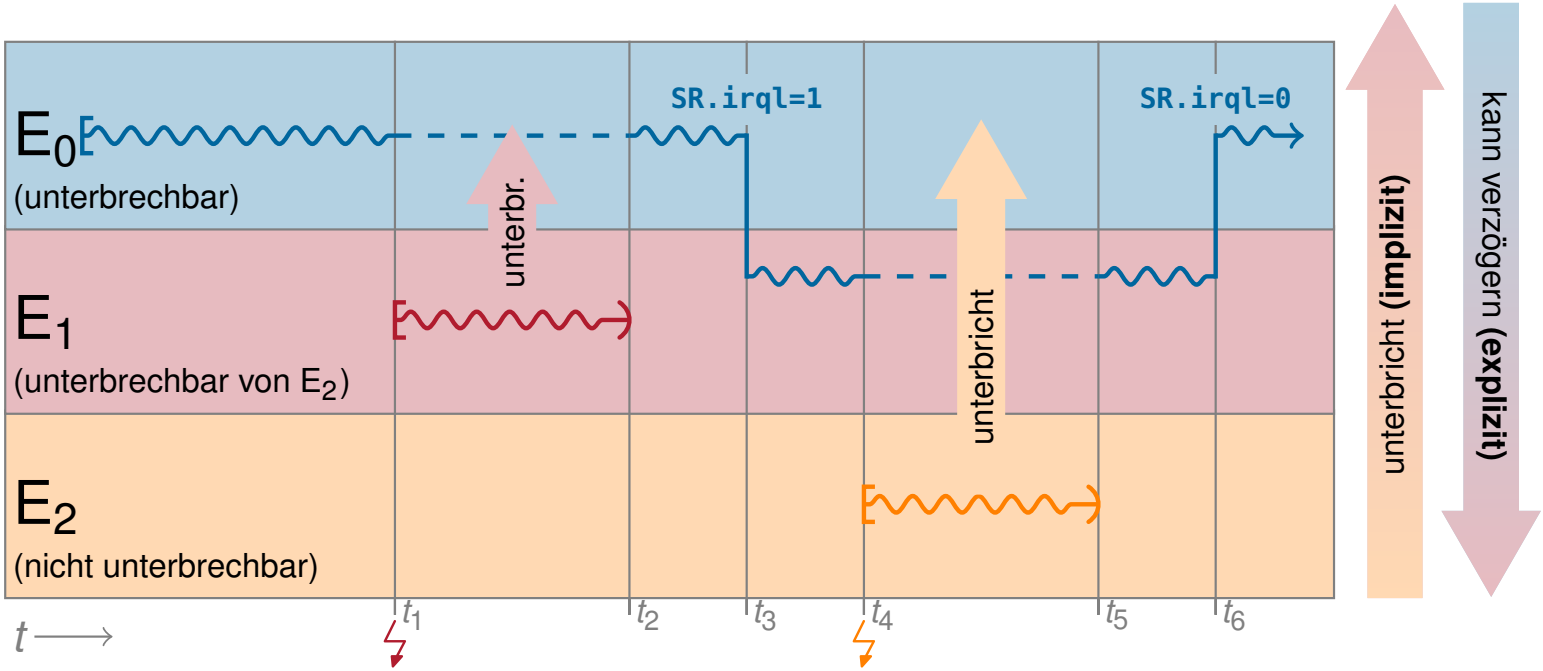
Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
  - Randbedingungen
  - Mutex, Implementierungsvarianten
  - Passives Warten
  - Semaphore
- Beispiel: Windows
  - Warteobjekte
  - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen

Prioritätsebenenmodell

(Wiederholung von VL 6)

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. sequentialisiert mit weiteren Kontrollflüssen von E_l



Prioritätsebenenmodell

(Wiederholung von VL 6)

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. sequentialisiert mit weiteren Kontrollflüssen von E_l

Das ist der Knackpunkt!

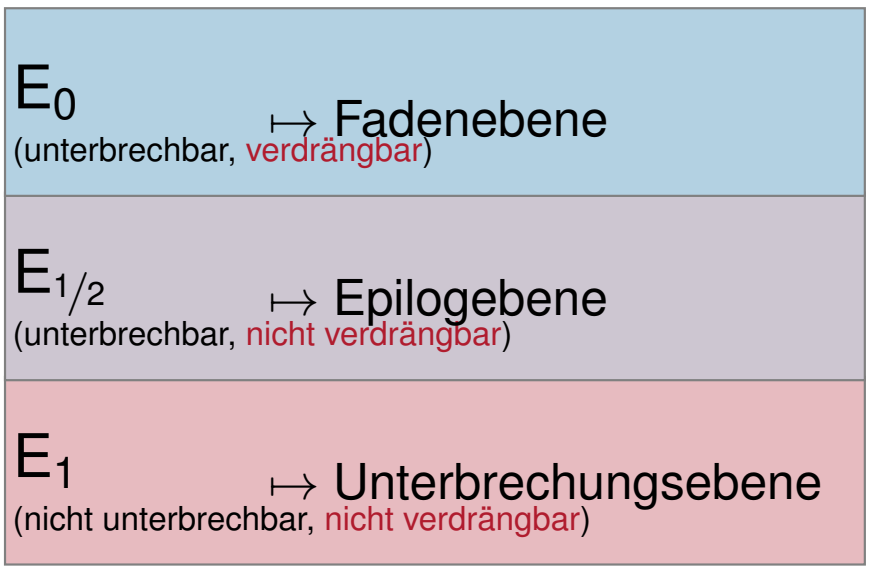
Mit der Unterstützung präemptiver Fäden können wir Annahme 3 nicht länger aufrechterhalten:

- keine *run-to-completion*–Semantik mehr
- Zugriff auf geteilten Zustand nicht mehr implizit sequentialisiert

Dies gilt für alle Ebenen, die Verdrängung (*Preemption*) oder echte Parallelität von Kontrollflüssen erlauben, also insbesondere die Anwendungsebene E_0.

Erweitertes Prioritätsebenenmodell

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. jederzeit verdrängt durch Kontrollflüsse von E_l (für $l = 0$)



Kontrollflüsse der E_0 (Fadenebene) sind **verdrängbar**.

Für die Konsistenzssicherung auf dieser Ebene brauchen wir zusätzliche **Mechanismen** zur **Fadensynchronisation**.

Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
  - Randbedingungen
  - Mutex, Implementierungsvarianten
  - Passives Warten
  - Semaphore
- Beispiel: Windows
  - Warteobjekte
  - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen

Fadensynchronisation: Annahmen

- Fäden können unvorhersehbar verdrängt werden
  - zu jedem beliebigen Zeitpunkt
  - von beliebigen anderen Fäden höherer, gleicher, oder niedrigerer Priorität ($\hookleftarrow$ Fortschrittsgarantie)
- Annahmen typisch für Arbeitsplatzrechner $\rightsquigarrow$ VL 9
  - *probabilistic, interactive, preemptive, online CPU scheduling*
  - andere Arten des Schedulings werden im Folgenden nicht betrachtet

Problematisch ist hier die Fortschrittsgarantie

Bei rein prioritätsgesteuertem Scheduling (Fäden innerhalb einer Prioritätsstufe werden sequentiell abgearbeitet) könnten wir das Ebenenmodell der Unterbrechungsbehandlung ($\rightsquigarrow$ VL 5) einfach auf Fadenprioritäten ausdehen und vergleichbaren Mechanismen (expliziter Ebenenwechsel, algorithmisch unter der Annahme von *run-to-completion*) synchronisieren.

- typisch für ereignisgesteuerte Echtzeitssysteme $\rightsquigarrow$ [EZS]
- in Windows/Linux: Bereich der Echtzeitprioritäten $\rightsquigarrow$ VL 9
- bei mehreren Kernen bleibt das Problem der echten Parallelität!

Fadensynchronisation: Überblick

- **Ziel:** aus Anwendersicht
  Koordinierung und Interaktion
  - Koordinierung des exklusiven Zugriffs auf wiederverwendbare Betriebsmittel (gegenseitiger Ausschluss) $\rightsquigarrow$ Mutex
  - Interaktion / Koordinierung von konsumierbaren Betriebsmitteln (Synchronisation) $\rightsquigarrow$ Semaphore

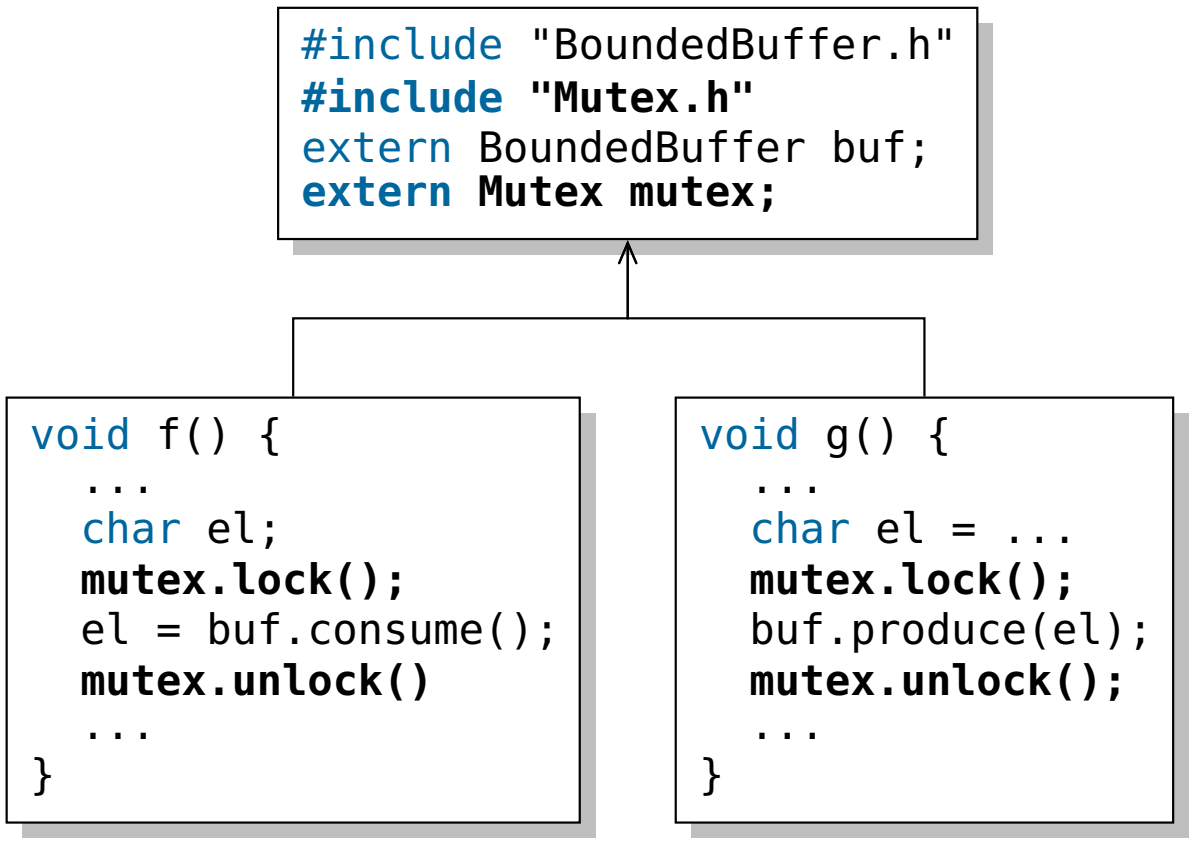
- **Implementierungsansatz:** für den BS-Entwickler
  Steuerung der CPU-Zuteilung an Fäden
  - Fäden werden zeitweise von der Zuteilung ausgenommen
  - „Warten” als BS-Konzept

Im Folgenden befassen wir uns mit der Perspektive der BS-Entwicklerin

Mutex

- **Mutex** $\mapsto$ Kurzform von **mu**tual **ex**clusion
  - Ursprung: Bezeichnername eines zweiwertigen Semaphor, eingesetzt für gegenseitigen Ausschluss [2]
  - allgemein: Algorithmus für die Sicherstellung von gegenseitigem Ausschluss in einem kritischen Gebiet
  - hier: Systemabstaktion `class Mutex`
- Schnittstelle
  - `void Mutex::lock()`
    - Betreten und Sperren des kritischen Gebiets
    - Faden kann blockieren
  - `void Mutex::unlock()`
    - Verlassen und Freigeben des kritischen Gebiets
- Korrektheitsbedingung
  - Es befindet sich maximal ein Faden im kritischen Gebiet
    - Für ausgeführte Operationen gilt: $\sum_{\texttt{lock()}} - \sum_{\texttt{unlock()}} \leq 1$

Mutex: Verwendung



Mutex mit aktivem Warten ($\mapsto$ *spin lock*)

- Implementierung rein auf der Benutzerebene
  - markiere Belegung in boolscher Variable (0 $\mapsto$ frei, 1 $\mapsto$ belegt)
  - warte in `lock()` aktiv, bis Variable 0 wird

```
// __sync_lock_test_and_set ist ein gcc builtin fuer
// (CPU-spezifisches) test-and-set (ab gcc 4.1)
class SpinningMutex {
  volatile int locked;
public:
  SpinningMutex() : locked (0) {}
  void lock() {
    while (__sync_lock_test_and_set(
            &locked, 1) == 1)
      ;
  }
  void unlock() {
    locked = 0;
  }
};
```

```
// g++-4.2 -O3
// -fomit-frame-pointer
lock:
    mov 0x4(%esp),%edx
l1: mov $0x1,%eax
    xchg %eax,(%edx)
    sub $0x1,%eax
    je l1
    repz ret
unlock:
    mov 0x4(%esp),%eax
    movl $0x0,(%eax)
    ret
```

Mutex mit aktivem Warten: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
    - unter der Vorraussetzung von Fortschrittsgarantie für alle Fäden
  - Synchronisation erfolgt ohne Beteiligung des Betriebssystems
    - keine Systemaufrufe erforderlich
- **Nachteile**
  - aktives Warten verschwendet viel CPU-Zeit
    - mindestens bis die Zeitscheibe abgelaufen ist
    - bei Zeitscheiben von 10–800 msec ganz erheblich!
    - Faden wird eventuell vom Scheduler „bestraft” (⇝ VL 9)

Fazit

Aktives Warten ist – wenn überhaupt – nur auf **Multiprozessormaschinen** eine Alternative.

Mutex mit harter Synchronisation

- Implementierung mit „harter Fadensynchronisation”
  - deaktiviere Verdrängbarkeit vor Betreten des kritischen Gebiets
    - neue Systemoperation: `forbid()`
  - reaktiviere Verdrängbarkeit nach Verlassen des kritischen Gebiets
    - neue Systemoperation: `permit()`

```
class HardMutex {
public:
  void lock() {
    forbid();      // schalte Multitasking ab
  }
  void unlock() {
    permit();      // schalte Multitasking wieder an
  }
};
```

In der Welt der Echtzeitsysteme steht dieses Verfahren hinter dem *non-preemptive critical section (NPCS) protocol* [6, EZS].

Mutex mit harter Synchronisation: Implementierung

- Implementierung durch den Scheduler, z. B. über
  - spezielle nicht verdrängbare Prioritätsklasse
    - OSEK OS / AUTOSAR OS: Ressource `RES_SCHED` [7]
  - eigene Prioritätsebene $E_{1/4}$ für den Scheduler
    - war faktisch so in AmigaOS realisiert
  - `resume()` schaltet einfach zum Aufrufer zurück
- oder ganz einfach durch Betreten der Epilogebene
  - Fadenumschaltung ist üblicherweise auf der Epilogebene angesiedelt
    - so lange ein Faden auf der Epilogebene ist kann er nicht verdrängt werden
    - Vorraussetzung: Kontrollflüsse der Epilogebene werden sequentialisiert
  - $\rightsquigarrow$ **Sequentialisierung auch mit Epilogen!**

```
void forbid() {
  enter();
}
void permit() {
  leave();
}
```

Mutex mit harter Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
  - einfach zu implementieren
- **Nachteile**
  - Breitbandwirkung
    - alle Fäden (und ggfs. sogar Epiloge!) werden pauschal verzögert
  - Prioritätsverletzung
    - „unbeteiligte" Kontrollflüsse mit höherer Priorität werden verzögert
  - prophylaktisches Verfahren
    - Nachteile werden in Kauf genommen, auch wenn die Wahrscheinlichkeit einer tatsächlichen Kollision sehr klein ist.

Fazit

Fadensynchronisation auf Epilogebene hat viele Nachteile. Sie ist nur auf Einprozessorsystemen für kurze, selten betretene kritische Gebiete geeignet – oder wenn sowieso mit Epilogen synchronisiert werden muss.

Passives Warten: Motivation

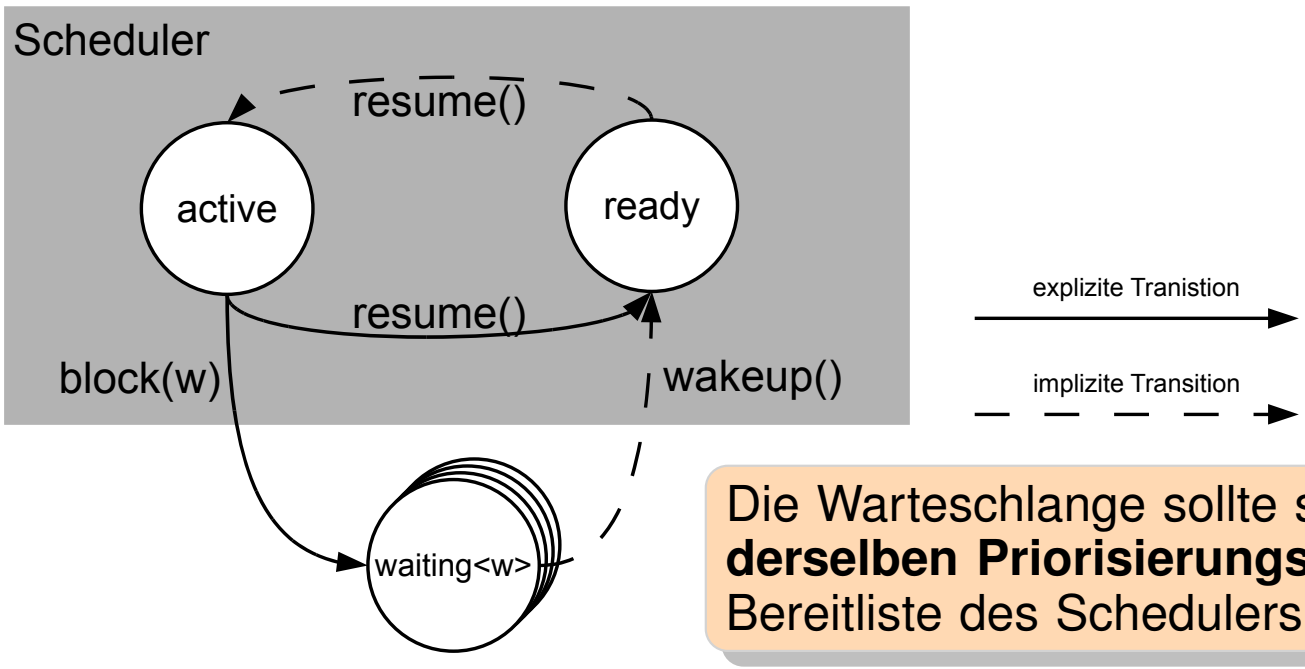
- Bisherige Mutex-Implementierungen sind nicht ideal
  - Mutex mit aktivem Warten
    $\leadsto$ Verschwendung von CPU-Zeit
  - Mutex mit harter Synchronisation
    $\leadsto$ grobgranular, proritätsverletzend

- **Besserer Ansatz:** Faden so lange von der CPU-Zuteilung ausschließen, wie der Mutex belegt ist.

- Erfordert neues BS-Konzept: **passives Warten**
  - Fäden können auf ein Ereignis „passiv warten”
    - passiv warten $\mapsto$ von CPU-Zuteilung ausgeschlossen sein
    - (Neuer) Fadenzustand: *wartend* (auf Ereignis)
  - Eintreffen des Ereignisses bewirkt Verlassen des Wartezustands
    - Faden wird in CPU-Zuteilung eingeschlossen
    - Anschließender Fadenzustand: *bereit*

Passives Warten: Implementierung

- Erforderliche Abstraktionen:
  - Operationen: `block()`, `wakeup()`
    - Betreten bzw. Verlassen des Wartezustands
  - Warteobjekt: `Waitingroom`
    - repräsentiert das Ereignis auf das gewartet wird
    - enthält üblicherweise eine Warteschlange der wartenden Fäden



Die Warteschlange sollte sinnvollerweise mit **derselben Priorisierungsstrategie** wie die Bereitliste des Schedulers verwaltet werden!

Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
  int locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    while (__sync_lock_test_and_set(&locked, 1) == 1)
        scheduler.block(*this);
  }
  void unlock() {
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
  }
};
```

Bei dieser Lösung gibt es noch ein Problem...

Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
  int volatile locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    mutex.lock();
    while (locked == 1)
        scheduler.block(*this);
    locked = 1;
    mutex.unlock();
  }
  void unlock() {
    mutex.lock();
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
    mutex.unlock();
  }
};
```

`lock()` und `unlock()` bilden ein eigenes kritisches Gebiet

Kann man dieses kritische Gebiet mit einem `Mutex` schützen?

Mutex mit passivem Warten: Implementierung

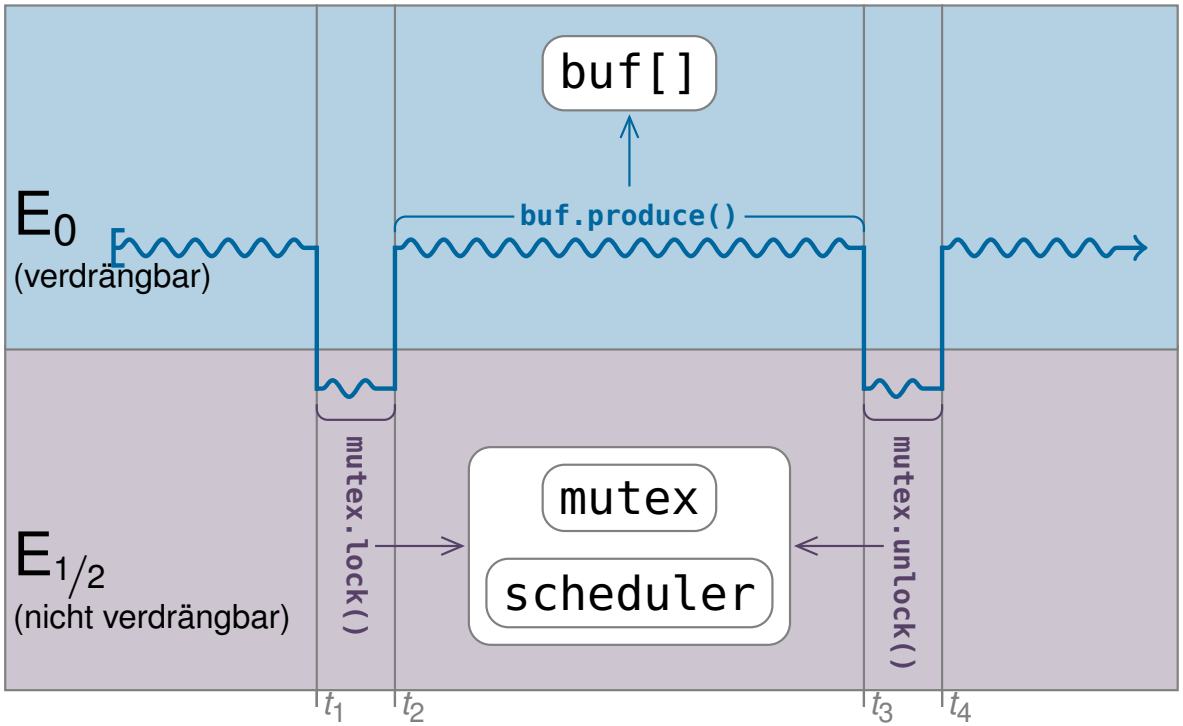
```
class WaitingMutex : public Waitingroom {
  int volatile locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    enter();
    while (locked == 1)
        scheduler.block(*this);
    locked = 1;
    leave();
  }
  void unlock() {
    enter();
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
    leave();
  }
};
```

Mit einem `HardMutex` ginge es!

Faktisch schützt man `lock()` und `unlock()` somit, wie hier dargestellt, auf **Epilogebene**.

Mutex mit passivem Warten: Fazit

- Mutex-Zustand liegt nun im Kern auf der Epilogebene
  - genauer: auf derselben Ebene wie der Scheduler Zustand
- Das ist ein allgemein verwendbares Prinzip
  - Implementierung der Synchronisationsmechanismen für E_0-Kontrollflüsse wird auf $E_{1/2}$ synchronisiert.



Noch besser wäre natürlich **weiche Synchronisation**.

Dazu mehr in [CS]!

Semaphore

- Semaphore ist das klassische Synchronisationsobjekt
  - Edgar W. Dijkstra, 1963 [3]
  - In vielen BS: Grundlage für alle Warte-/Synchronisationsobjekte
  - Für uns: Semaphore $\mapsto$ Warteobjekt + Zähler
- Zwei Standardoperationen (mit jeweils diversen Namen [2–4])
  - `prolaag()`, **`P()`**, `wait()`, `down()`, `acquire()`, `pend()`
    - wenn zähler $>$ 0 vermindere Zähler
    - wenn zähler $\leq$ 0 warte bis Zähler $>$ 0 und probiere es noch einmal
  - `verhoog()`, **`V()`**, `signal()`, `up()`, `release()`, `post()`
    - erhöhe Zähler
    - wenn Zähler $=$ 1 wecke gegebenenfalls wartenden Faden
- Es gibt vielfältigste Varianten

Implementierung der Standardvariante erfolgt in der Übung!

Semaphore: Verwendung

- Semantik der Semaphore eignet sich besonders für die Implementierung von Erzeuger/Verbraucher-Szenarien
  - Also für den geordneten Zugriff auf konsumierbare Betriebsmittel
    - Zeichen von der Tastatur
    - Signale, die auf Fadenebene weiterverarbeitet werden sollen
    - ...
  - Interner Zähler repräsentiert die Anzahl der Ressourcen
    - Erzeuger ruft `V()` auf für jedes erzeugte Element.
    - Verbraucher ruft `P()` auf, um ein Element zu konsumieren
      $\rightsquigarrow$ wartet gegebenenfalls.

Beachte!

- `P()` kann auf Fadenebene blockieren, `V()` blockiert jedoch nie!
- Als Erzeuger kommt daher auch ein Kontrollfluss auf Epilogebene oder Unterbrechungsebene in Frage. (Entsprechende Synchronisation des internen Semaphorzustands vorrausgesetzt.)

Semaphore vs. Mutex: Einordnung

- Mutex wird „klassisch“ als binärer Semaphor bezeichnet [2]
  - Mutex $\mapsto$ Semaphore mit initialem Zählerwert 1
  - `lock()` $\mapsto$ `P()`, `unlock()` $\mapsto$ `V()`
- Die Semantik ist (heute) jedoch i. a. deutlich strenger:
  - Ein belegter Mutex hat (implizit oder explizit) einen Besitzer.
    - Nur dieser Besitzer darf `unlock()` aufrufen.
    - Muteximplementierungen in z. B. Linux oder Windows überprüfen dies.
  - Ein Mutex kann (üblicherweise) auch rekursiv belegt werden
    - Interner Zähler: *Derselbe* Faden kann mehrfach `lock()` aufrufen; nach der entsprechenden Anzahl von `unlock()`-Aufrufen ist der Mutex frei
    - Eine Semaphore kann hingegen von *jedem* Faden verändert werden.

Semaphore als Basis aller Dinge?

In vielen BS ist Semaphore die Grundabstraktion für Fadensynchronisation. Sie wird deshalb in der Literatur oft als (notwendige) Implementierungbasis für Mutex, Bedingungsvariable, Leser-Schreiber-Sperre etc. angesehen.

Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
  - Randbedingungen
  - Mutex, Implementierungsvarianten
  - Passives Warten
  - Semaphore
- Beispiel: Windows
  - Warteobjekte
  - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen

Fadensynchronisation unter Windows

- Windows treibt die Idee der Warteobjekte sehr weit
  - **Jedes** Kernobjekt ist (auch) ein Synchronisationsobjekt!
    - explizite Synchronisationsobjekte: Event, Mutex, Timer, Semaphore, ...
    - implizite Synchronisationsobjekte: File, Socket, Thread, Prozess, ...
  - Semantik des Wartens hängt vom Objekt ab
    - Faden wartet auf „signalisiert"-Zustand
    - Zustand wird gegebenenfalls durch erfolgreiches Warten geändert
- Einheitliche, mächtige Systemschnittstelle für alle Objekttypen
  - Jedes Kernobjekt wird repräsentiert durch ein `HANDLE`
  - `WaitForSingleObject(hObject, dwMillisec)`
    - Wartet auf ein Synchronisationsobjekt mit Timeout
  - `WaitForMultipleObjects(nCount, hObjects[], bWaitAll, dwMillisec)`
    - Wartet auf einen Vektor von Synchronisationsobjekten mit Timeout (und/oder Warten, je nach `bWaitAll` = true/false)

Synchronisationsobjekte unter Windows

Objekt	ist signalisiert, wenn	erfolgreiches warten bewirkt
Event	Ändern des Zustands erfolgt explizit durch `SetEvent()` / `ResetEvent()`	zurückgesetzen des Events (nur bei AutoReset-Events)
Mutex	der Mutex verfügbar ist	Besitzname des Mutex
Semaphore	der Zähler der Semaphore > 0 ist	vermindern des Wertes der Semphore um 1
Waitable Timer	ein bestimmter Zeitpunkt erreicht wurde	zurücksetzen des Timers (nur bei AutoReset-Timern)
Change Notification	eine bestimmte Änderung im Dateisystem stattfand	keine Änderung des Zustands
Console Input	Eingabedaten zur Verfügung stehen	keine Änderung, solange Zeichen verfügbar sind
Process	der Prozess terminiert ist	keine Änderung des Zustands
Thread	der Thread terminiert ist	keine Änderung des Zustands
File	eine asynchrone Dateioperation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Dateioperation begonnen wird
Serial device	Daten verfügbar sind / Dateioperation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Operation begonnen wird
NamedPipe	eine asynchrone Operation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Dateioperation begonnen wird
Socket	eine asynchrone Operation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Operation begonnen wird
Job	Prozesse des Jobs terminiert sind	keine Änderung des Zustands
...		

Kosten der Fadensynchronisation

- Synchronisationsobjekte werden im Kern verwaltet
  - interne Datenstrukturen (Scheduler) $\rightsquigarrow$ Schutz
  - interne Synchronisation $\rightsquigarrow$ Konsistenz
- Das kann ihre Verwendung sehr teuer machen
  - für jede Zustandsänderung muss in den Kern gewechselt werden
  - Benutzer-/Kernmodus-Transitionen sind sehr aufwändig
  - Bei IA32 kommen schnell einige tausend Takte zusammen!
- Bei kurzen kritischen Gebieten mit geringer Wettstreitigkeit (*Contention*) schlägt dies besonders ins Gewicht
  - Die benötigte Zeit, um den Mutex zu akquirieren und freizugeben ist oft ein Vielfaches der Zeit, die das kritische Gebiet belegt ist.
  - Eine tatsächliche Konkurrenzsituation (Faden will in ein bereits belegtes kritisches Gebiet) tritt nur selten auf.

Kosten der Fadensynchronisation

- **Ansatz:** Mutex soweit wie möglich im Benutzermodus verwalten
  - Minimieren der Kosten im Normalfall
    - Normalfall $\mapsto$ kritisches Gebiet ist frei
    - Speziallfall $\mapsto$ kritisches Gebiet ist belegt
- Einführen eines *fast path* für den Normalfall
  - Test, Belegung, und Freigabe erfolgt im Benutzermodus
    - Konsistenz wird algorithmisch / durch atomare CPU-Befehle sichergestellt
  - Warten erfolgt im Kernmodus
    - für den Übergang in den passiven Wartezustand wird der Kern benötigt
  - weitere Optimierung für Multiprozessormaschinen
    - vor dem passiven Warten für begrenzte Zeit aktiv warten
      $\rightsquigarrow$ hohe Wahrscheinlichkeit, dass das kritische Gebiet vorher frei wird

Windows: CRITICAL_SECTION

- Struktur für einen *fast mutex* im Benutzermodus [8, 9]
  - verwendet intern ein Event (Kernobjekt), falls gewartet werden muss
  - Event wird „lazy“ (erst bei Bedarf) erzeugt
- Eigene Systemschnittstelle
  - `EnterCriticalSection(pCS)` / `TryEnterCriticalSection(pCS)`
    - k.G. belegen (blockierend) / versuchen zu belegen (nicht-blockierend)
  - `LeaveCriticalSection(pCS)`
    - kritisches Gebiet verlassen
  - `SetCriticalSectionSpinCount(pCS, dwSpinCount)`
    - Anzahl der Versuche für aktives Warten festlegen (nur auf MP-Systemen)

```
typedef struct _CRITICAL_SECTION {
  LONG LockCount;       // Anzahl der wartenden Threads (-1 wenn frei)
  LONG RecursionCount; // Anzahl der erfolgreichen EnterXXX-Aufrufe
  DWORD OwningThread;  // des Besitzers (OwningThread)
  HANDLE LockEvent;     // internes Warteobjekt, bei Bedarf erzeugt
  ULONG SpinCount;      // Auf MP-Systemem: Anzahl der busy-wait
                        // Versuche, bevor im Kern passiv gewartet wird
} CRITICAL_SECTION, *PCRITICAL_SECTION;
```

Windows: CRITICAL_SECTION

- Struktur für einen *fast mutex* im Benutzermodus [8, 9]
  - verwendet intern ein Event (Kernobjekt), falls gewartet werden muss
  - Event wird „lazy“ (erst bei Bedarf) erzeugt
- Eigene Systemschnittstelle
  - `EnterCriticalSection(pCS)` / `TryEnterCriticalSection(pCS)`
    - k.G. belegen (blockierend) / versuchen zu belegen (nicht-blockierend)
  - `LeaveCriticalSection(pCS)`
    - kritisches Gebiet verlassen
  - `SetCriticalSectionSpinCount(pCS, dwSpinCount)`
    - Anzahl der Versuche für aktives Warten festlegen (nur auf MP-Systemen)

```
typedef struct _CRITICAL_SECTION {
  LONG LockCount;       // Anzahl der wartenden Threads (-1 wenn frei)
  LONG RecursionCount; // Anzahl der erfolgreichen EnterXXX-Aufrufe
  DWORD OwningThread;  // des Besitzers (OwningThread)
  HANDLE LockEvent;    // internes
  ULONG SpinCount;     // Auf MP-Sy
                       // Versuche,
} CRITICAL_SECTION, *PCRITICAL_SECT
```

Unter Linux gibt es ab Kernel 2.6 mit Futexes (*F*ast *u*ser-mode mu*texes*) ein vergleichbares, noch deutlich mächtigeres Konzept. [1, 5]

Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
  - Randbedingungen
  - Mutex, Implementierungsvarianten
  - Passives Warten
  - Semaphore
- Beispiel: Windows
  - Warteobjekte
  - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen

Zusammenfassung: Fadensynchronisation

- Programmfäden können jederzeit verdrängt werden
  - präemptives, probabilistisches Multitasking
  - keine run-to-completion–Semantik
  - Zugriff auf geteilten Zustand muss gesondert synchronisiert werden
- Fadensynchronisation: Ein Markt der Möglichkeiten
  - Mutex für gegenseitigen Ausschluss
  - Semaphore für Erzeuger-/Verbraucher-Szenarien
  - viele weitere Abstraktionen möglich: Leser-/Schreiber–Sperren, Vektorsemaphoren, Bedingungsvariablen, Timeouts, ...
- Grundlage ist ein BS-Konzept für passives Warten
  - Fundamentale Eigenschaft von Fäden: Sie können warten
  - aktives Warten und harte Fadensynchronisation sind (nur) in Ausnahmefällen sinnvoll

Referenzen

Ulrich Depper. *Futexes are tricky*. Techn. Ber. (Version 1.5). Red Hat Inc., Aug. 2009. URL: https://akkadia.org/drepper/futex.pdf (besucht am 06. 01. 2011).

Edsger Wybe Dijkstra. *Cooperating Sequential Processes*. Techn. Ber. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996). Eindhoven, The Netherlands: Technische Universiteit Eindhoven, 1965. URL: https://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF.

Edsger Wybe Dijkstra. „Multiprogrammering en de X8“. circulated privately. Juni 1963. URL: https://www.cs.utexas.edu/users/EWD/ewd00xx/EWD57.PDF (besucht am 06. 01. 2011).

Per Brinch Hansen. *Betriebssysteme*. München: Carl Hanser Verlag, 1977. ISBN: 3-446-12105-6.

Matthew Kirkwood Hubertus Franke Rusty Russell. „Fuss, futexes and furwocks: Fast Userlevel Locking in Linux“. In: *Proceedings of the Ottawa Linux Symposium* (Ottawa, OT, Canada, 26. Juni 2002–29. Juni 2002). Hrsg. von Andrew J. Hutton, Stephanie Donovan und C. Craig Ross. Juni 2002, S. 479–495.

Referenzen (Forts.)

Aloysius K. L. Mok. „Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment“. Diss. Cambridge, MA, USA: Massachusetts Institute of Technology, MIT, Mai 1983.

OSEK/VDX Group. *Operating System Specification 2.2.3*. Techn. Ber. OSEK/VDX Group, Feb. 2005.

Matt Pietrek und Russ Osterlund. „Break Free of Code Deadlocks in Critical Sections Under Windows“. In: *MSDN Magazine* 18 (12 Dez. 2003). ISSN: 1528-4859. URL: http://msdn.microsoft.com/en-us/magazine/cc164040.aspx (besucht am 06. 01. 2011).

Jeffrey Richter. *Windows via C/C++*. 5. Aufl. Microsoft Press, 2007. ISBN: 978-0735624245.

Wolfgang Schröder-Preikschat. *Concurrent Systems*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_CS.

Peter Ulbrich. *Echtzeitsysteme*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_EZS.

Betriebssysteme (BS)

VL 12 – Gerätetreiber

Volkmar Sieh / **Daniel Lohmann**

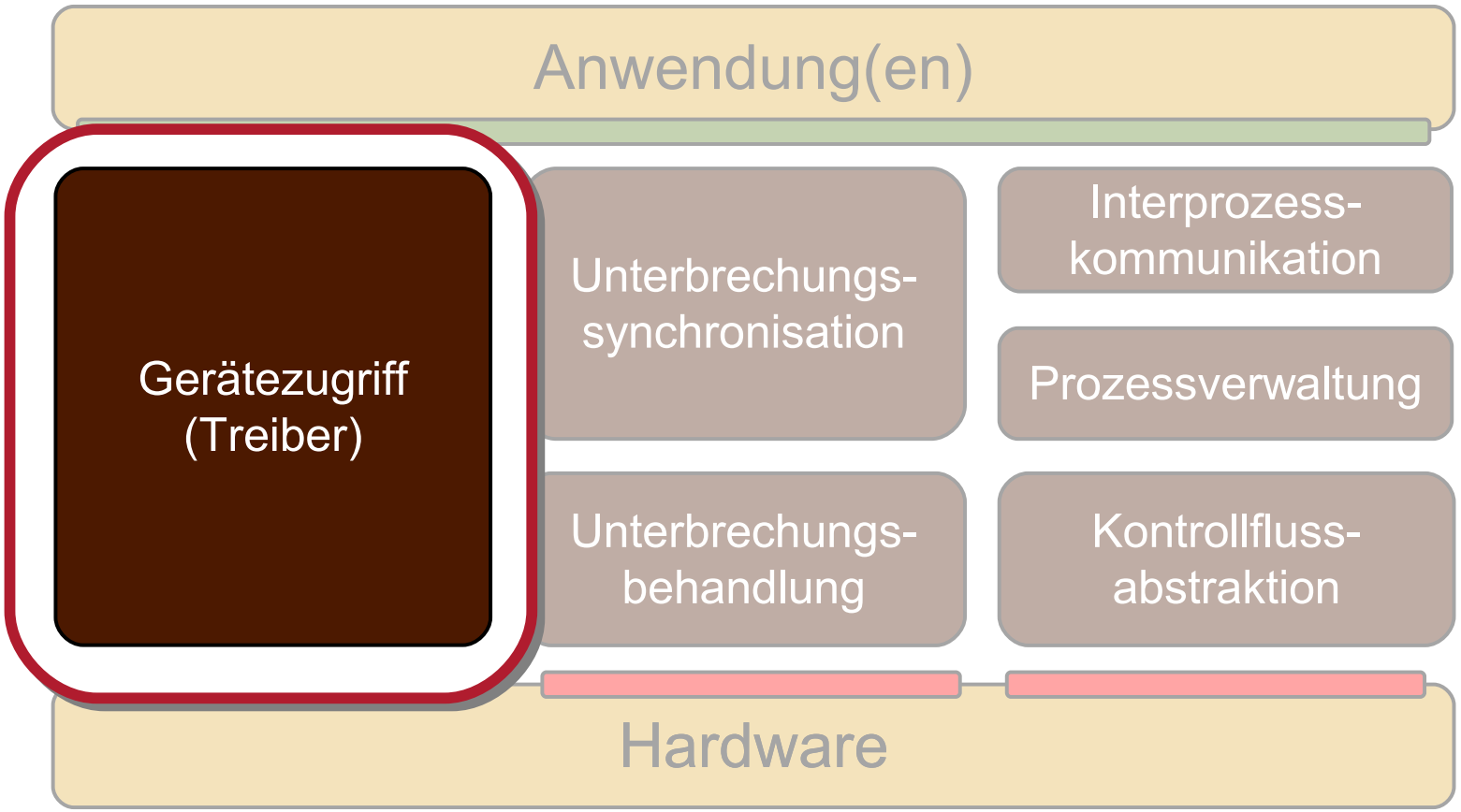
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 24. Januar 2024

`https://sys.cs.fau.de/lehre/ws23/bs`

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

Einordnung
Anforderungen an das BS
Struktur des E/A-Systems
Zusammenfassung

Agenda

Einordnung

- Bedeutung von Gerätetreibern

Anforderungen an das BS

Struktur des E/A-Systems

Zusammenfassung

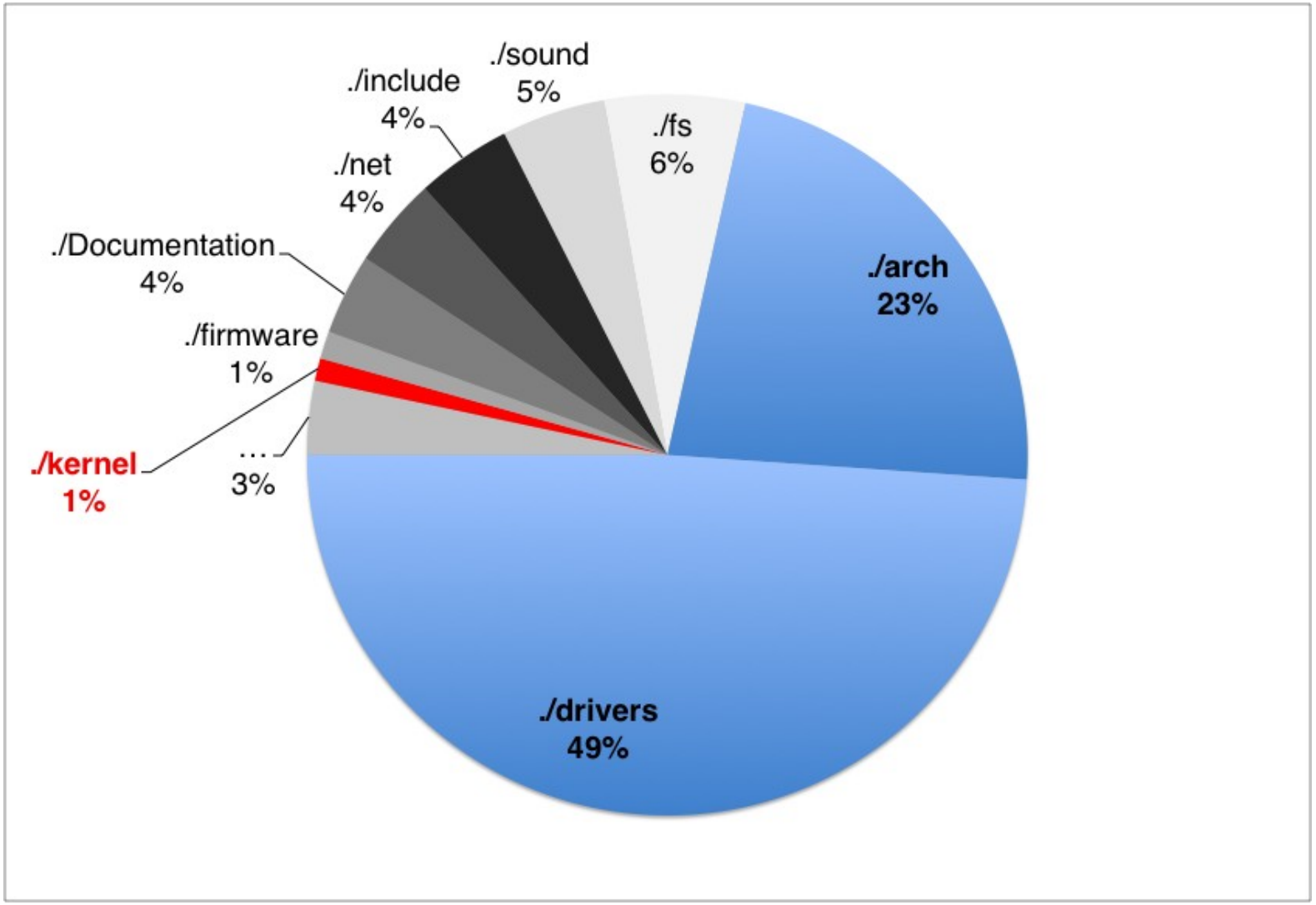
Bedeutung von Gerätetreibern (1)

Anteil Treiber-Sourcen in Linux-5.10.3:

```
linux-5.10.3> du -skh * | sort -n
...
  4.2M  mm
  6.2M  lib
 11M    kernel
 34M    net
 41M    sound
 43M    fs
 43M    tools
 47M    include
 53M    Documentation
134M    arch
670M    drivers
```

Bedeutung von Gerätetreibern (1)

- Anteil an Treibercode in Linux 3.2.1



Bedeutung von Gerätetreibern (2)

- in Linux (3.2.1) ist der Treibercode (ohne ./arch) etwa **50 mal so groß** wie der Code des Kernels
  - und wächst rasant!
    - bei V2.6.32 waren es "nur" 25 mal mehr
    - bei V2.6.11 waren es "nur" 10 mal mehr
  - Windows unterstützt noch deutlich mehr Geräte ...
- Treiberunterstützung ist für die Akzeptanz eines Betriebssystems ein **entscheidender Faktor**
  - warum sonst wäre Linux weiter verbreitet als andere freie UNIXe?
- in Gerätetreibern steckt eine erhebliche Arbeitsleistung

➔ der Entwurf des E/A Subsystems erfordert viel Geschick

- möglichst viele wiederverwendbare Funktionen in eine **Treiber-Infrastruktur** verlagern
- klare Vorgaben bzgl. Treiberstruktur, -verhalten und -schnittstellen, d.h. ein **Treibermodell**

Agenda

Einordnung

Anforderungen an das BS

- Einheitlicher Zugriff
- Spezifischer Zugriff

Struktur des E/A-Systems

Zusammenfassung

Anforderungen an Betriebssysteme

- Ressourcenschonender Umgang mit Geräten
  - schnell arbeiten
  - Energie sparen
  - Speicher, *Ports* und *Interrupt*-Vektoren sparen
  - Aktivierung und Deaktivierung zur Laufzeit
  - Generische *Power Management* Schnittstelle
- Einheitlicher Zugriffsmechanismus
  - **minimaler Satz von Operationen** für verschiedene Gerätetypen
  - **mächtige Operationen** für vielfältige Typen von Anwendungen
- auch gerätespezifische Zugriffsfunktionen

Linux – einheitlicher Zugriff (1)

```
echo "Hallo, Welt" > /dev/ttyS0
```

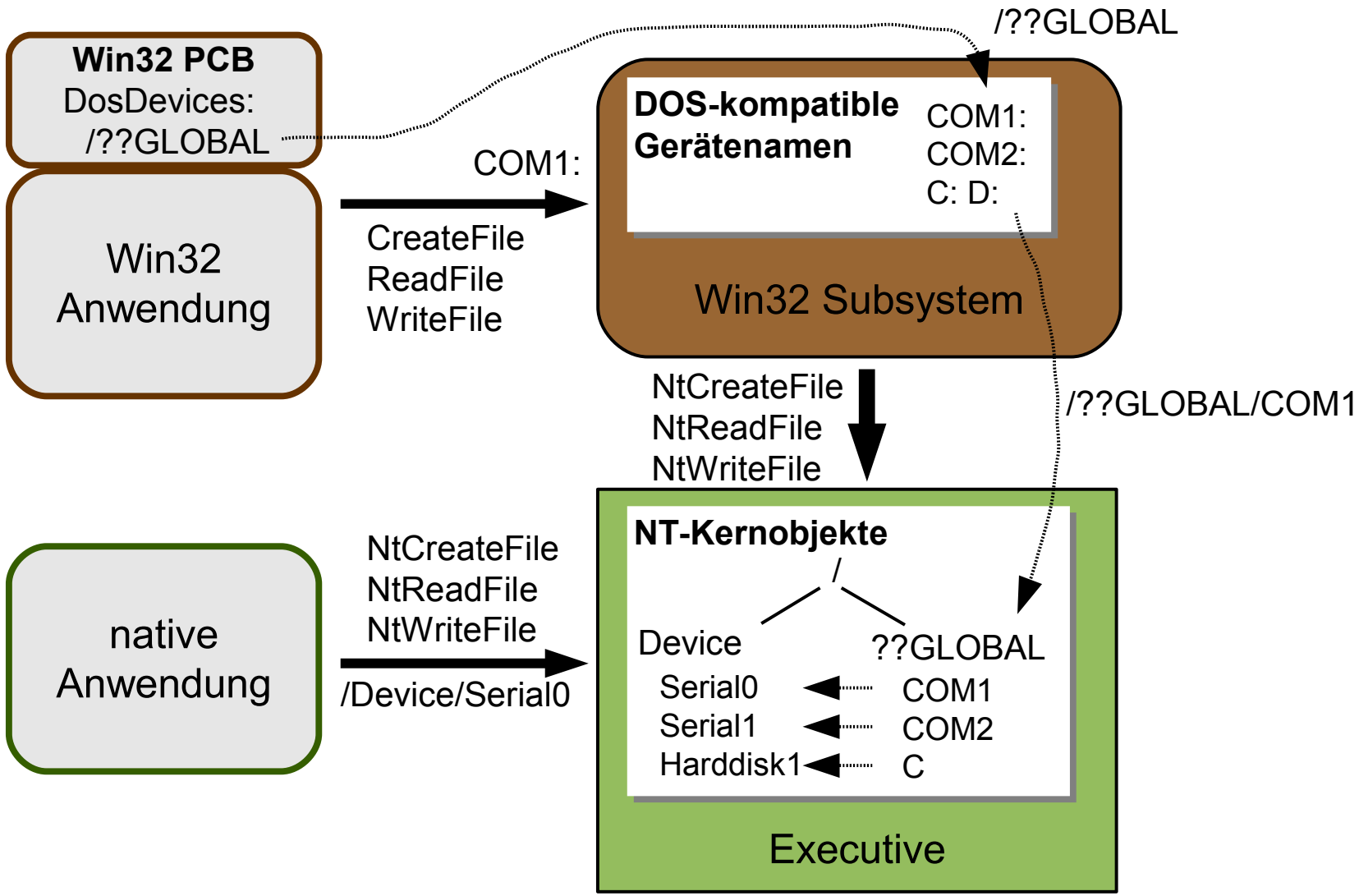
- Geräte sind über Namen im Dateisystem ansprechbar
- Vorteile:
  - Systemaufrufe für Dateizugriff (`open`, `read`, `write`, `close`) können auch für sonstige E/A verwendet werden
  - Zugriffsrechte können über die Mechanismen des Dateisystems gesteuert werden
  - Anwendungen sehen keinen Unterschied zwischen Dateien und "Gerätedateien"
- Probleme:
  - blockorientierte Geräte müssen in Byte-Strom verwandelt werden
  - manche Geräte lassen sich nur schwer in dieses Schema pressen
    - Beispiel: 3D Graphikkarte

Linux – einheitlicher Zugriff (2)

- blockierende Ein-/Ausgabe (Normalfall)
  - `read`: Prozess blockiert bis die angeforderten Daten da sind
  - `write`: Prozess blockiert bis Schreiben möglich ist
- nicht-blockierende Ein-/Ausgabe
  - `open`/`read`/`write` mit dem Zusatz-Flag `O_NONBLOCK`
  - statt zu blockieren kehren `read` und `write` so mit `-EAGAIN` zurück
  - der Aufrufer kann/muss die Operation später wiederholen
- nebenläufige Ein-/Ausgabe
  - neu: `aio_(read|write|...)` (POSIX 1003.1-2003)
  - indirekt mittels Kindprozess (`fork`/`join`)
  - `select`, `poll` Systemaufrufe

Windows – einheitlicher Zugriff (1)

- Geräte sind Kern-Objekte der Executive



Windows – einheitlicher Zugriff (2)

- synchrone oder asynchrone Ein-/Ausgabe

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

NULL: synchrones Lesen

```
BOOL GetOverlappedResult(
  HANDLE hFile,
  LPOVERLAPPED lpOverlapped,
  LPDWORD lpNumberOfBytesTransferred,
  BOOL bWait
);
```

true: auf Ende warten
false: Status erfragen

- weitere Möglichkeiten:
  - E/A mit *Timeout*
  - WaitForMultipleObjects – warten auf 1–N Kernobjekte
    - Datei-Handles, Semaphore, Mutex, Thread-Handle, ...
  - I/O Completion Ports
    - Aktivierung eines wartenden Threads nach I/O Operation

Linux – gerätespez. Funktionen (1)

- spezielle Geräteeigenschaften werden (klassisch) über **ioctl** angesprochen:

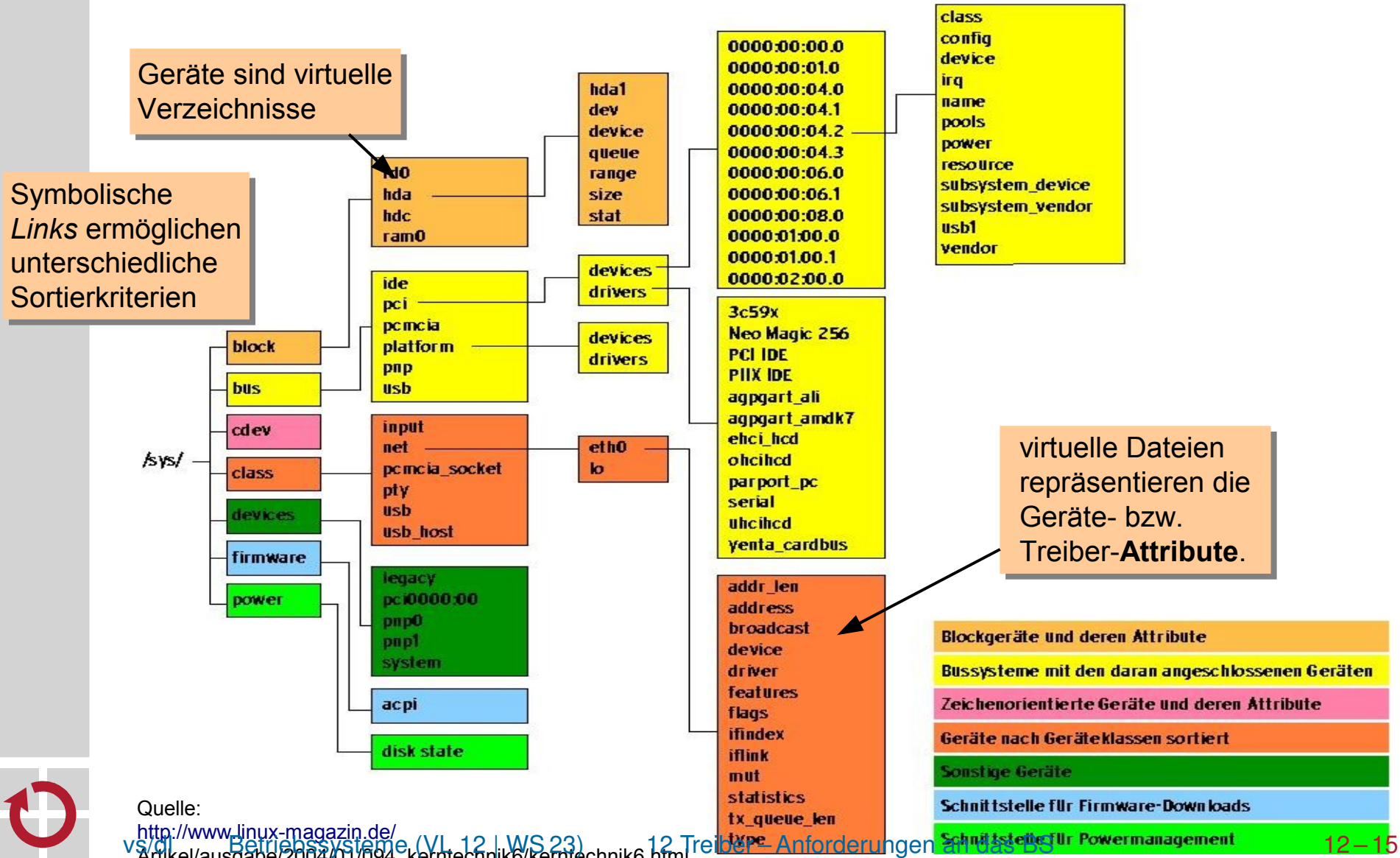
```
IOCTL(2)          Linux Programmer's Manual          IOCTL(2)

NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

       int ioctl(int d, int request, ...);
```

- Schnittstelle generisch und Semantik gerätespezifisch:

```
CONFORMING TO
       No single standard. Arguments, returns, and semantics of
       ioctl(2) vary according to the device driver in question
       (the call is used as a catch-all for operations that
       don't cleanly fit the Unix stream I/O model). The ioctl
       function call appeared in Version 7 AT&T Unix.
```

Linux – gerätespez. Funktionen (2)

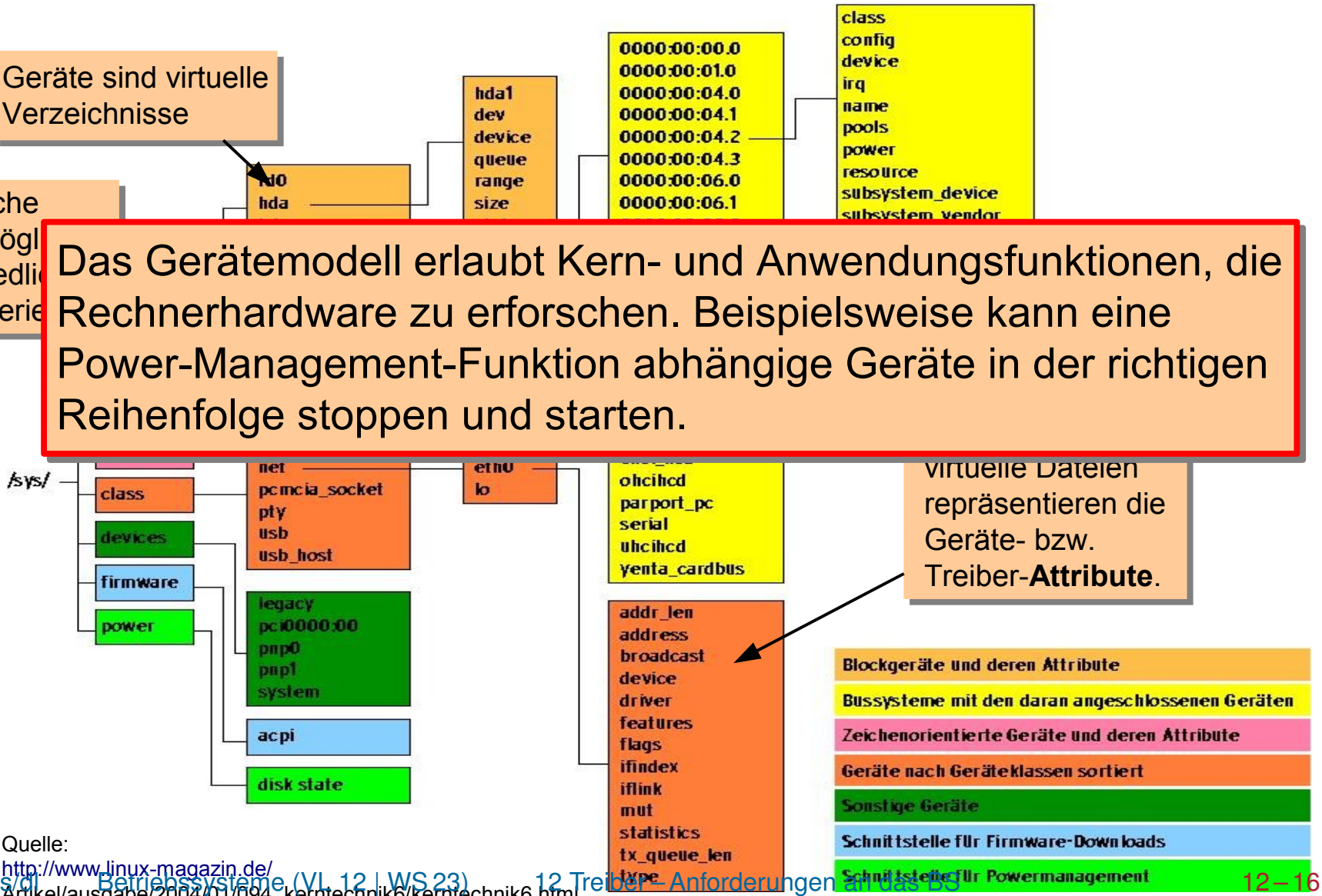
Linux 2.6 – das **Gerätemodell im sys-Dateisystem**



Quelle:
http://www.linux-magazin.de/Artikel/ausgabe/2004/01/094_kerntechnik6/kerntechnik6.html

Linux – gerätespez. Funktionen (2)

Linux 2.6 – das **Gerätemodell im sys-Dateisystem**



Quelle:
http://www.linux-magazin.de/Artikel/ausgabe/2004/01/094_kerntechnik6/kerntechnik6.html

Windows – gerätespez. Funktionen

- **DeviceIoControl** entspricht dem UNIX ioctl:

```
BOOL DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

Kommunikation über untypisierte Puffer direkt mit dem Treiber

auch asynchron möglich

- und was sonst?
  - alle Geräte und Treiber werden durch Kern-Objekte repräsentiert
    - spezielle Systemaufrufe gestatten das Erforschen dieses Namensraums
  - statische Konfigurierung erfolgt über die Registry
  - dynamische Konfigurierung erfolgt z.B. über WMI
    - *Windows Management Instrumentation*

Agenda

Einordnung

Anforderungen an das BS

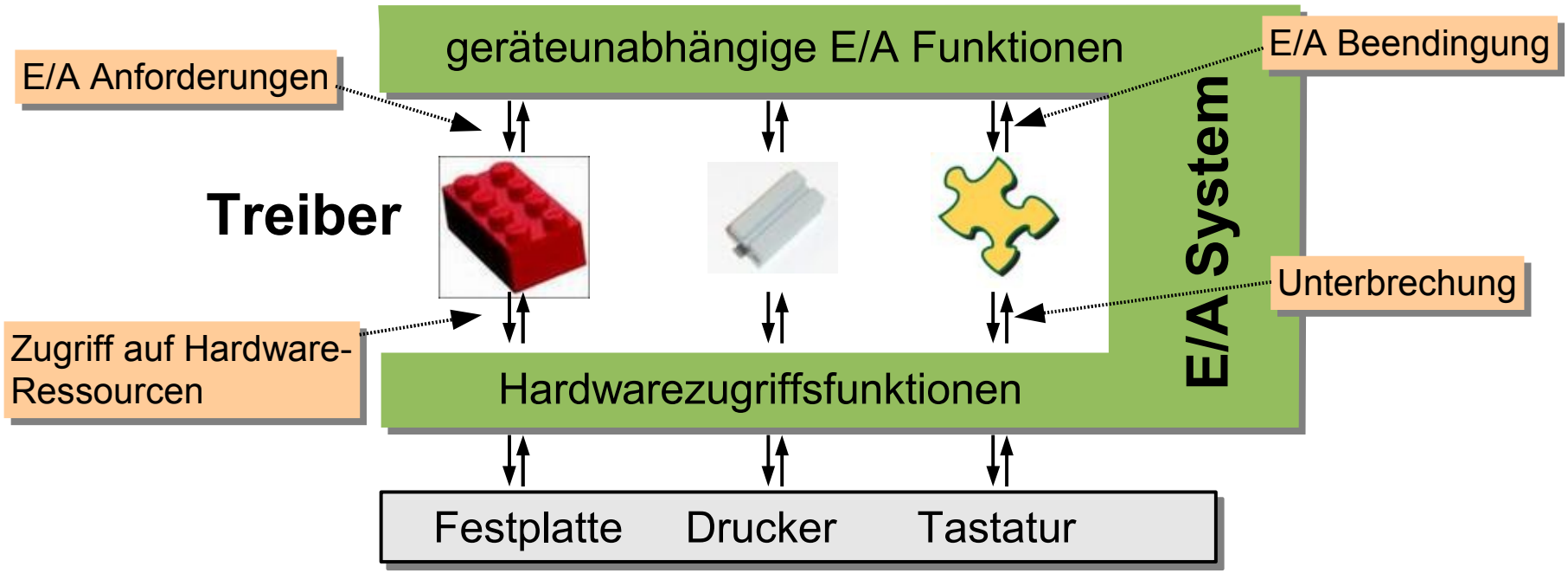
Struktur des E/A-Systems

- Treibermodell
- Linux
- Windows

Zusammenfassung

Struktur des E/A Systems (1)

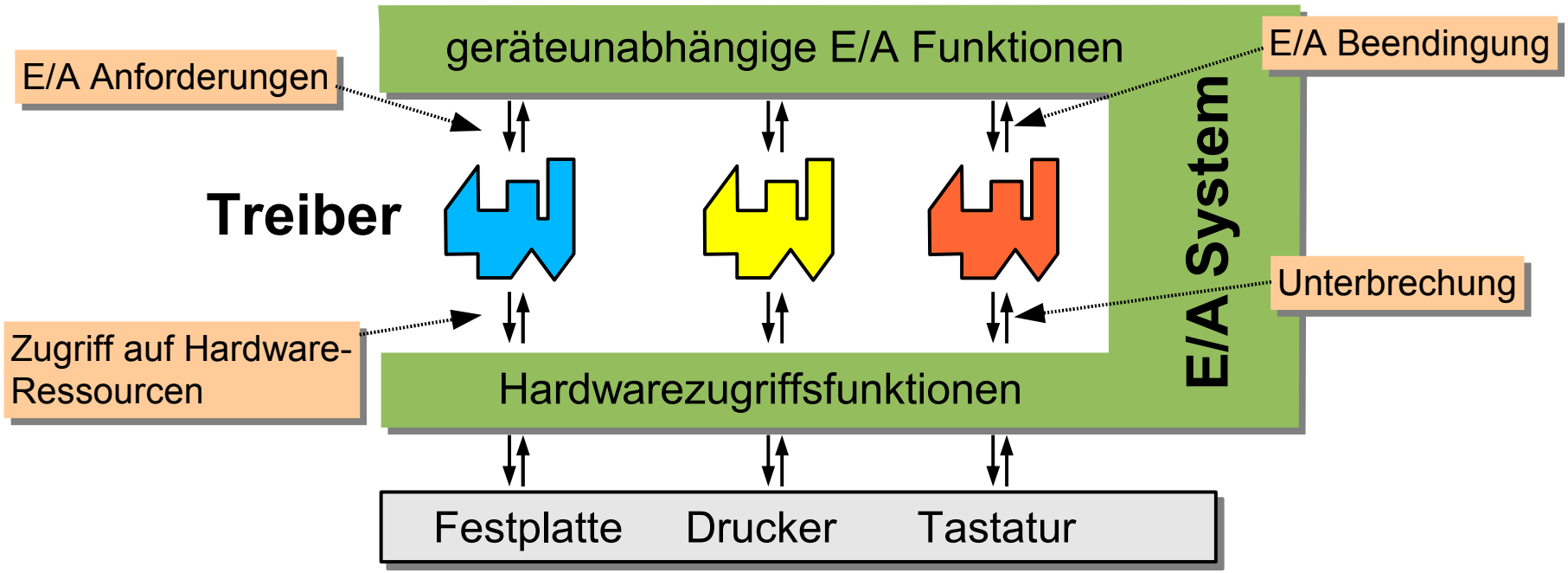
- Treiber mit unterschiedlicher Schnittstelle ...



- erlauben die volle Ausnutzung aller Geräteeigenschaften
- erfordern eine Erweiterung des E/A Systems für jeden Treiber
  - enormer Aufwand bei der heutigen Gerätevielfalt
  - unrealistisch, da erst das BS und dann die Treiber entstehen

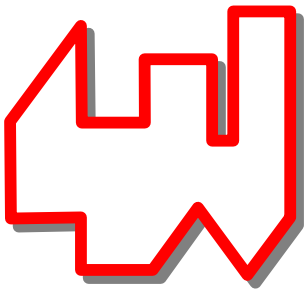
Struktur des E/A Systems (2)

- Treiber mit uniformer Schnittstelle ...



- ermöglichen ein (dynamisch) erweiterbares E/A System
- erlauben flexibles "Stapeln" von Gerätetreibern
  - virtuelle Geräte
  - Filter

Das Treibermodell umfasst ...



"detaillierte Vorgaben für die Treiber-Entwicklung"

- die Liste der erwarteten Treiber-Funktionen
- Festlegung optionaler und obligatorischer Funktionen
- die Funktionen, die ein Treiber nutzen darf
- Interaktionsprotokolle
- Synchronisationsschema und Funktionen

- Festlegung von **Treiberklassen** falls mehrere Schnittstellentypen unvermeidbar sind

Anforderungen an Gerätetreiber

- Zuordnung zu Gerätedateien erlauben
- Verwaltung mehrerer Geräteinstanzen
- Operationen:
  - Hardware-Erkennung
  - Initialisierung und Beendigung
  - Lesen und Schreiben von Daten
    - ggf. auch *Scatter/Gather*
  - Steueroperationen und Gerätestatus
    - z.B. über ioctl oder virtuelles Dateisystem
  - Energieverwaltung
- intern zu bewältigen:
  - Synchronisation
  - Pufferung
  - Anforderung benötigter Systemressourcen

Linux – Treibergerüst: Registrierung

```
MODULE_AUTHOR("B.S. Student");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Dummy Treiber.");
MODULE_SUPPORTED_DEVICE("none");

static struct file_operations fops;
// ... Initialisierung von fops (Funktionszeiger)

static int __init mod_init(void){
  if(register_chrdev(240,"DummyDriver",&fops)==0)
    return 0; // Treiber erfolgreich angemeldet
  return -EIO; // Anmeldung beim Kernel fehlgeschlagen
}

static void __exit mod_exit(void){
  unregister_chrdev(240,"DummyDriver");
}

module_init( mod_init );
module_exit( mod_exit );
```

Metainformation, anzufragen mit 'modinfo'

Registrierung für das char-Device mit der **Major-Number 240**

mod_init und mod_exit werden beim Laden bzw. Entladen ausgeführt.

Linux – Treibergerüst: Operationen

```
static char hello_world[]="Hello World\n";

static int dummy_open(struct inode *geraete_datei,
  struct file *instanz) {
    printk("driver_open called\n"); return 0;
}

static int dummy_close(struct inode *geraete_datei,
  struct file *instanz) {
    printk("driver_close called\n"); return 0;
}

static ssize_t dummy_read(struct file *instanz,
  char *user, size_t count, loff_t *offset ) {
    int not_copied, to_copy;
    to_copy = strlen(hello_world)+1;
    if( to_copy > count ) to_copy = count;
    not_copied=copy_to_user(user,hello_world,to_copy);
    return to_copy-not_copied;
}

static struct file_operations fops = {
  .owner  =THIS_MODULE,
  .open   =dummy_open,
  .release=dummy_close,
  .read   =dummy_read,
};
```

die Treiberoperationen entsprechen den normalen Dateioperationen

in diesem Beispiel machen open und close nur Debugging-Ausgaben

mit **copy_to_user** und **copy_from_user** kann man Daten zwischen Kern- und Benutzer-adressraum austauschen

es gibt noch wesentlich mehr Operationen, sie sind jedoch größtenteils optional

Linux – Treibergerüst: Operationen

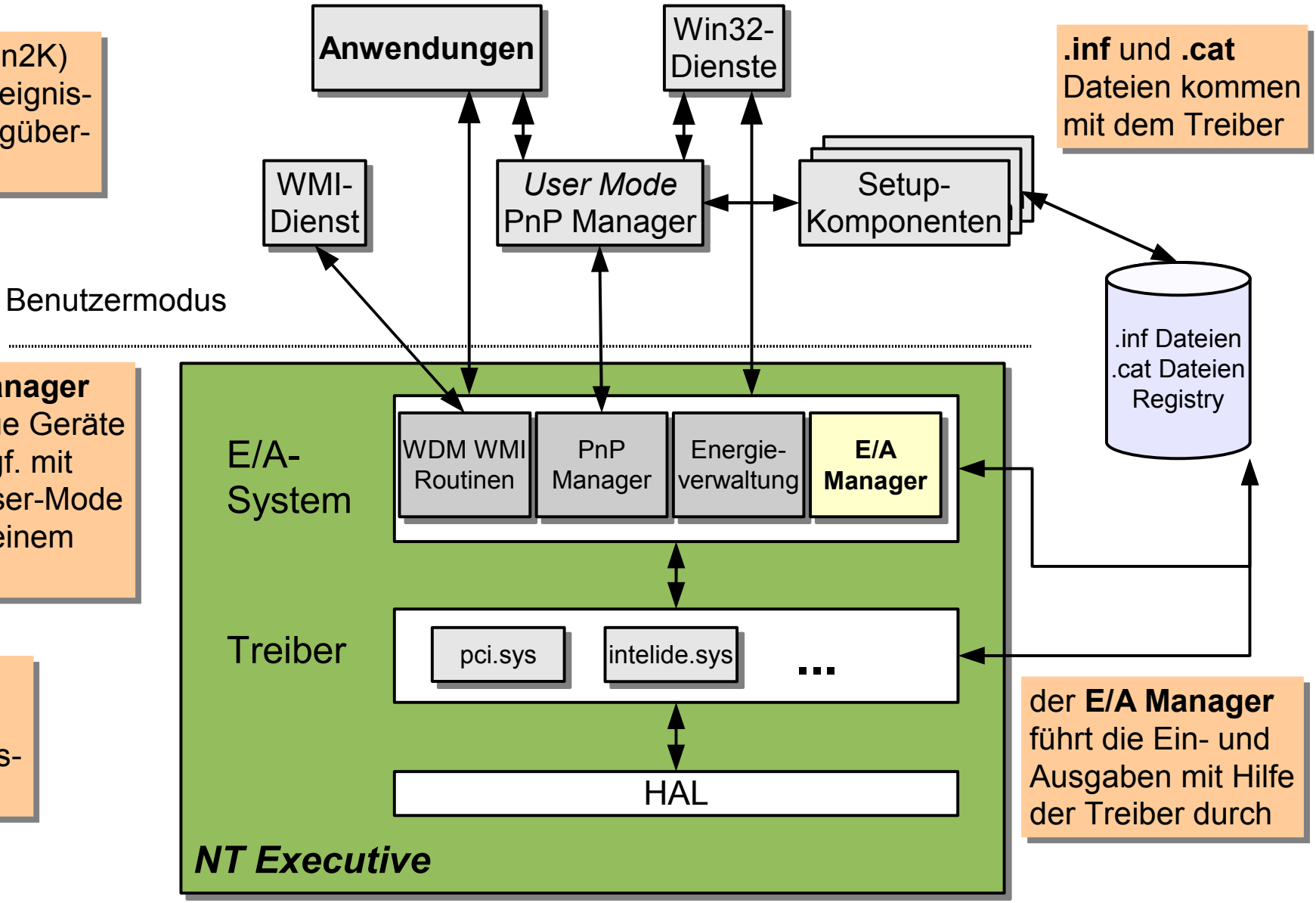
```
// Struktur zur Einbindung des Treibers in das virtuelle Dateisystem
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long,
        unsigned long, unsigned long, unsigned long);
};
```

Linux - Treiber-Infrastruktur

- Ressourcen reservieren
  - Speicher, Ports, IRQ-Vektoren, DMA Kanäle
- Hardwarezugriff
  - Ports und Speicherblöcke lesen und schreiben
- Speicher dynamisch anfordern
- Blockieren und Wecken von Prozessen im Treiber
  - waitqueue
- Interrupt-Handler anbinden
  - low-level
  - Tasklets für länger dauernde Aktivitäten
- Spezielle APIs für verschiedene Treiberklassen
  - Zeichenorientierte Geräte, Blockgeräte, USB-Geräte, Netzwerktreiber
- Einbindung in das `proc` oder `sys` Dateisystem

Windows – E/A System

WMI (ab Win2K) dient der Ereignis- und Leistungüberwachung



.inf und **.cat** Dateien kommen mit dem Treiber

der **PnP Manager** erkennt neue Geräte und fragt ggf. mit Hilfe des User-Mode Teils nach einem Treiber.

HAL ist die Hardware-Abstraktions-schicht

der **E/A Manager** führt die Ein- und Ausgaben mit Hilfe der Treiber durch

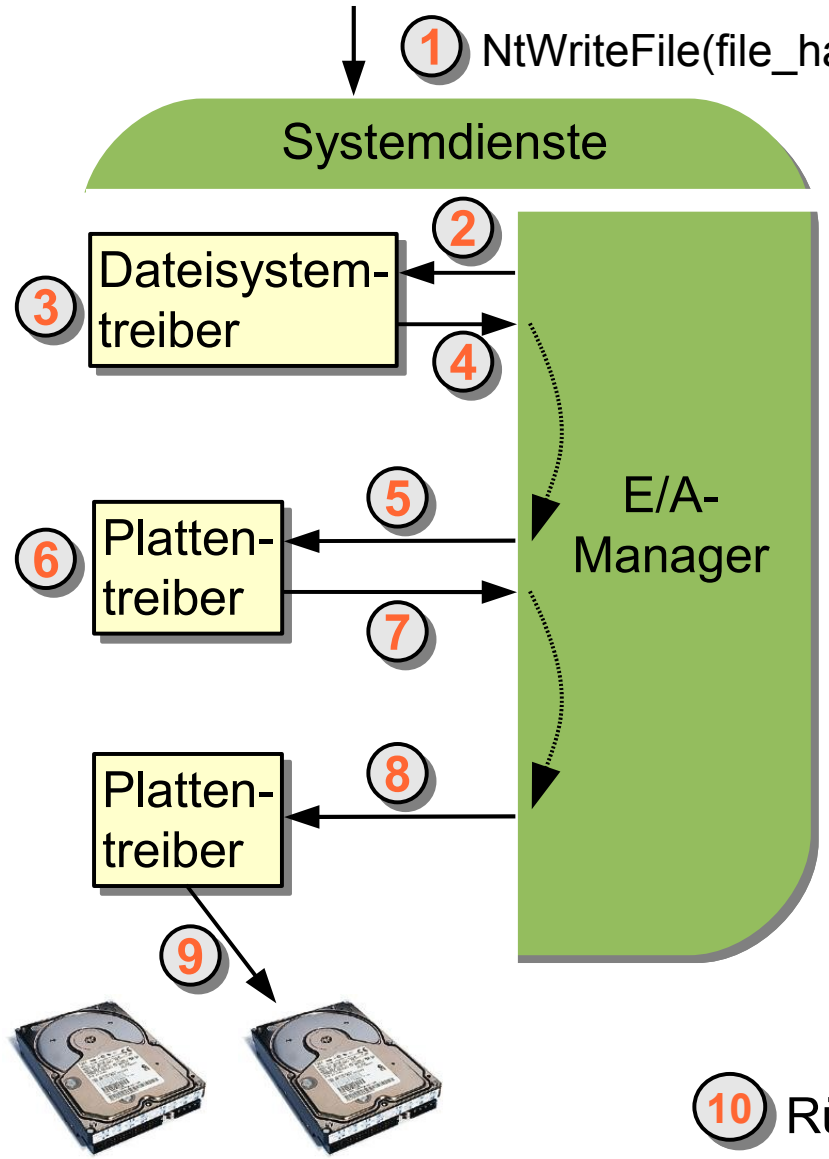
Windows – Treiberstruktur

Das E/A-System steuert den Treiber mit Hilfe der ...

- Initialisierungsroutine/Entladeroutine
  - wird nach/vor dem Laden/Entladen des Treibers ausgeführt
- Routine zum Hinzufügen von Geräten
  - PnP Manager hat ein neues Gerät für den Treiber
- "Verteilerroutinen"
  - Öffnen, Schließen, Lesen, Schreiben und gerätespezifische Oper.
- Interrupt Service Routine
  - wird von der zentralen Interrupt-Verteilungsroutine aufgerufen
- DPC-Routine
  - "Epilog" der Unterbrechungsbehandlung
- E/A-Komplettierungs- und -Abbruchroutine
  - Informationen über den Ausgang weitergeleiteter E/A-Aufträge

...

Windows – typischer E/A-Ablauf



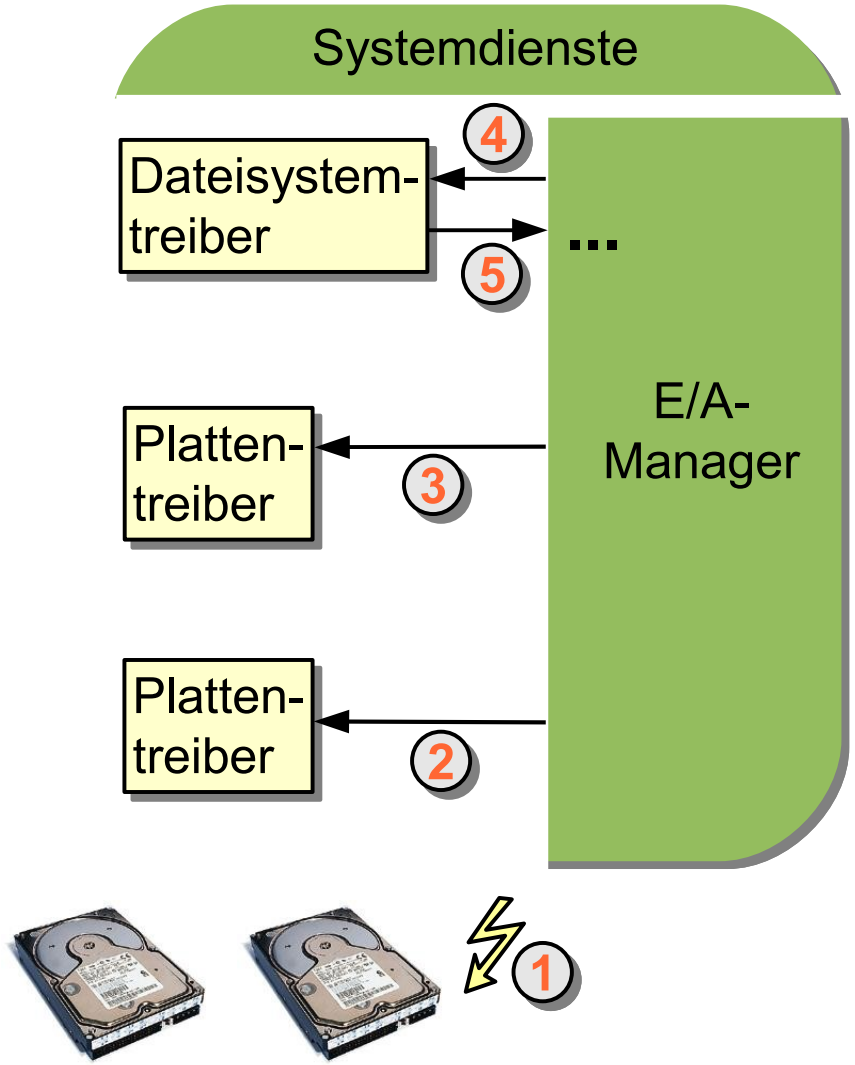
1. über das Dateiobjekt wird das Dateisystem und der Treiber gefunden
2. Daten an bestimmten Byte-Offset in Datei schreiben
3. Position auf Datenträger berechnen
4. E/A Auftrag weitergeben
5. Daten an best. Byte Offset auf Datenträger schreiben
6. Position in Plattennr. und Offset umrechen
7. E/A Auftrag weitergeben
8. Daten an best. Byte Offset auf Platte 2 schreiben
9. Phys. Block berechnen und Operation initiieren

10 Rückkehr zum Anwendungsprozess!

Windows – typischer E/A-Ablauf

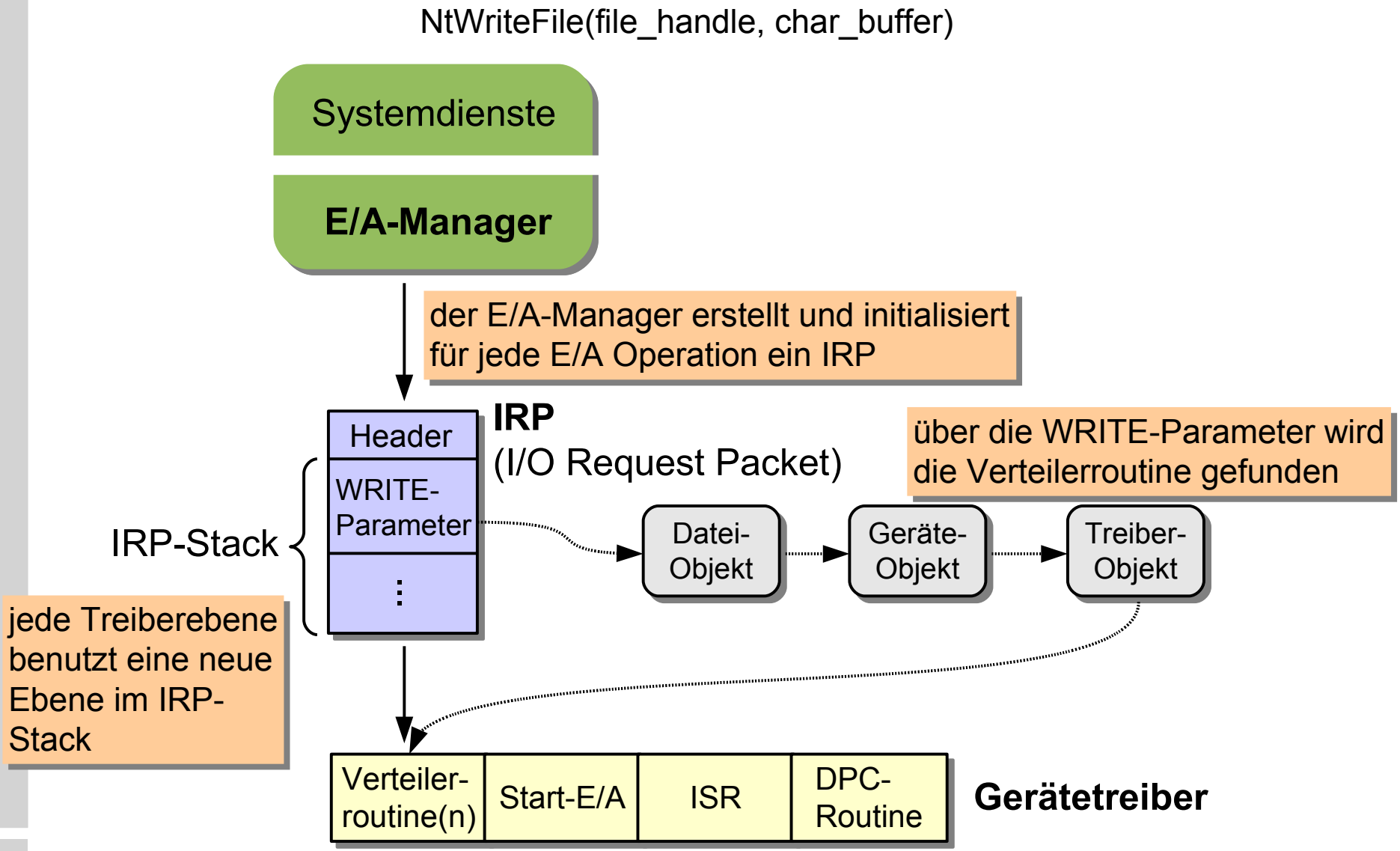
... Fortsetzung (nachdem die Platte fertig geworden ist)

1. Plattencontroller signalisiert per Unterbrechung den Abschluss der Operation
2. Aufruf der ISR bzw. des DPC
3. Aufruf der Komplettierungs-routine
4. Aufruf der Komplettierungs-routine
5. weiterer (Teil-)Auftrag an den Datenträgertreiber



Wo merkt sich das System den Zustand einer E/A-Operation?

Windows – E/A-Anforderungspaket



Agenda

Einordnung
Anforderungen an das BS
Struktur des E/A-Systems
Zusammenfassung

Zusammenfassung

- ein guter Entwurf des E/A Subsystems ist enorm wichtig
  - E/A-Schnittstelle
  - Treibermodell
  - Treiberinfrastruktur
  - Schnittstellen sollten lange stabil bleiben
- Ziel ist die Aufwandsminimierung bei der Treibererstellung
- Windows besitzt ein ausgereiftes E/A System
  - "alles ist ein Kern-Objekt"
  - asynchrone E/A Operationen sind die Basis
- Linux zieht rasant nach
  - "alles ist eine Datei"
  - sysfs und asynchrone E/A sind relativ neu (seit 2.6)

Betriebssysteme (BS)

VL 13 – Interprozesskommunikation

Volkmar Sieh / **Daniel Lohmann**

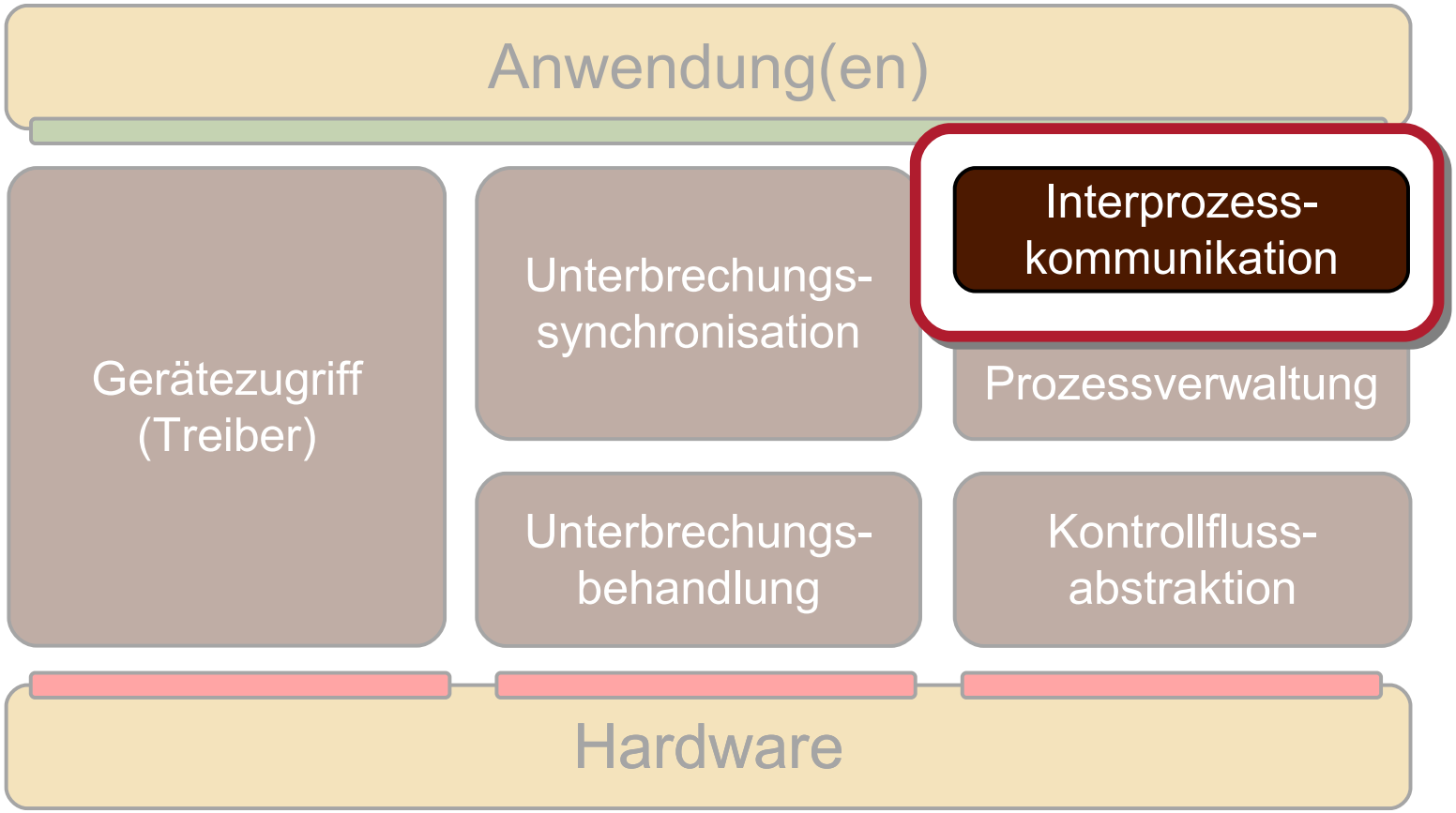
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 31. Januar 2024

https://sys.cs.fau.de/lehre/ws23/bs

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

Einordnung
IPC über Speicher
IPC über Nachrichten
Basisabstraktionen
Trennung der Belange mit AOP
Zusammenfassung

Agenda

Einordnung

Kommunikation und Synchronisation

IPC über Speicher

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung

Kommunikation und Synchronisation

- ... sind durch das Kausalprinzip immer verbunden:

> Wenn **A** eine Information von **B** benötigt, um weiterzuarbeiten, muss **A** solange ***warten***, bis **B** die Information bereitstellt.

- ➔ nachrichtenbasierte Kommunikation impliziert Synchronisation (z.B. bei `send()` und `receive()`)
- ➔ Synchronisationsprimitiven eignen sich als Basis für die Implementierung von Kommunikationsprimitiven (z.B. Semaphore)

Agenda

Einordnung

IPC über Speicher

- Monitore
- Pfadausdrücke
- Thread-Safety-Analyse

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung

IPC über gemeinsamen Speicher

- Anwendungsfälle/Voraussetzungen
  - ungeschütztes System (alle Prozesse im selben Adressraum)
  - System mit sprachbasiertem Speicherschutz
  - Kommunikation zwischen Fäden im selben Adressraum
  - gemeinsamer Speicher mit Hilfe des BS und einer MMU (z.B. UNIX System V shared memory)
  - gemeinsamer Kern-Adressraum von isolierten Prozessen
- positive Eigenschaften:
  - atomare Speicherzugriffe erfordern keine zusätzliche Synchronisation
  - schnell: kein Kopieren
  - einfache IPC Anwendungen leicht zu realisieren
  - unsynchronisierte Kommunikationsbeziehungen möglich
  - M:N Kommunikation leicht möglich

Semaphore – einfache Interaktionen

- gegenseitiger Ausschluss

```
// gem. Speicher
Semaphore mutex(1);
SomeType  shared;
```

```
void process_1() {
  mutex.wait();
  shared.access();
  mutex.signal();
}
```

```
void process_2() {
  mutex.wait();
  shared.access();
  mutex.signal();
}
```

- einseitige Synchronisation

```
// gem. Speicher
Semaphore elem(0);
SomeQueue shared;
```

```
void producer() {
  shared.put();
  elem.signal();
}
```

```
void consumer() {
  elem.wait();
  shared.get();
}
```

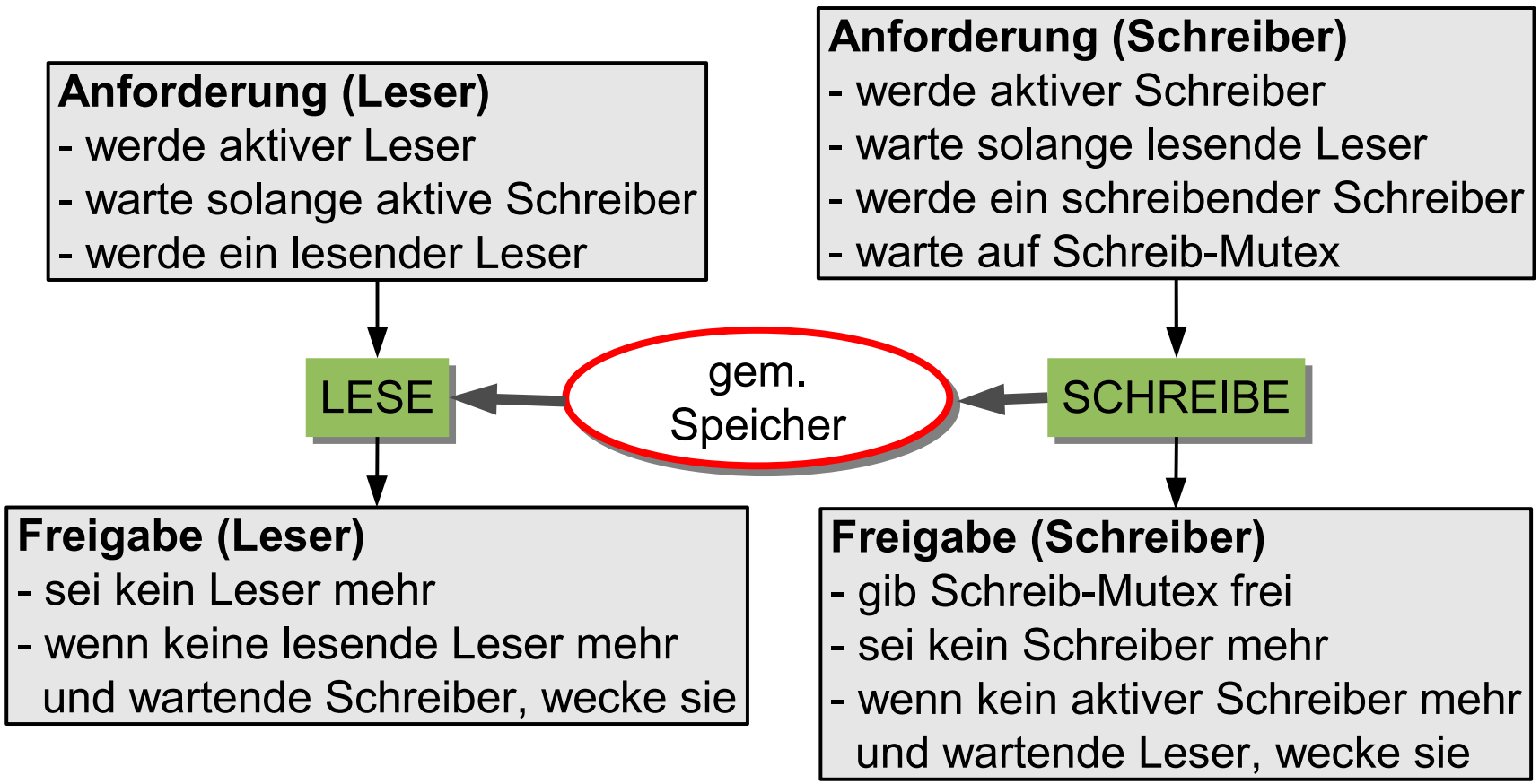
- betriebsmittelorientierte Synchronisation

```
// gem. Speicher
Semaphore     resource(N); // N>1
SomeResource  shared;
```

sonst wie beim gegenseitigen Ausschluss

Semaphore – komplexe Interaktionen

- Leser/Schreiber-Problem
  - Schreiber benötigen den Speicher exklusiv
  - mehrere Leser können gleichzeitig arbeiten



Semaphore – Leser/Schreiber-Problem

```
// Anforderung (Leser)
mutex.p();
ar++; // aktive Leser
if (aw==0) {
  rr++; // lesende Leser
  read.v();
}
mutex.v();
read.p();
```

```
// Anforderung (Schreiber)
mutex.p();
aw++; // aktive Schreiber
if (rr==0) {
  ww++; // schreibende S.
  write.v();
}
mutex.v();
write.p();
w_mutex.p();
```

```
// Freigabe (Leser)
mutex.p();
ar--; rr--;
while (rr==0 && ww<aw) {
  ww++;
  write.v();
}
mutex.v();
```

```
// Freigabe (Schreiber)
w_mutex.v();
mutex.p();
aw--; ww--;
while (aw==0 && rr<ar) {
  rr++;
  read.v();
}
mutex.v();
```

Semaphore – Diskussion

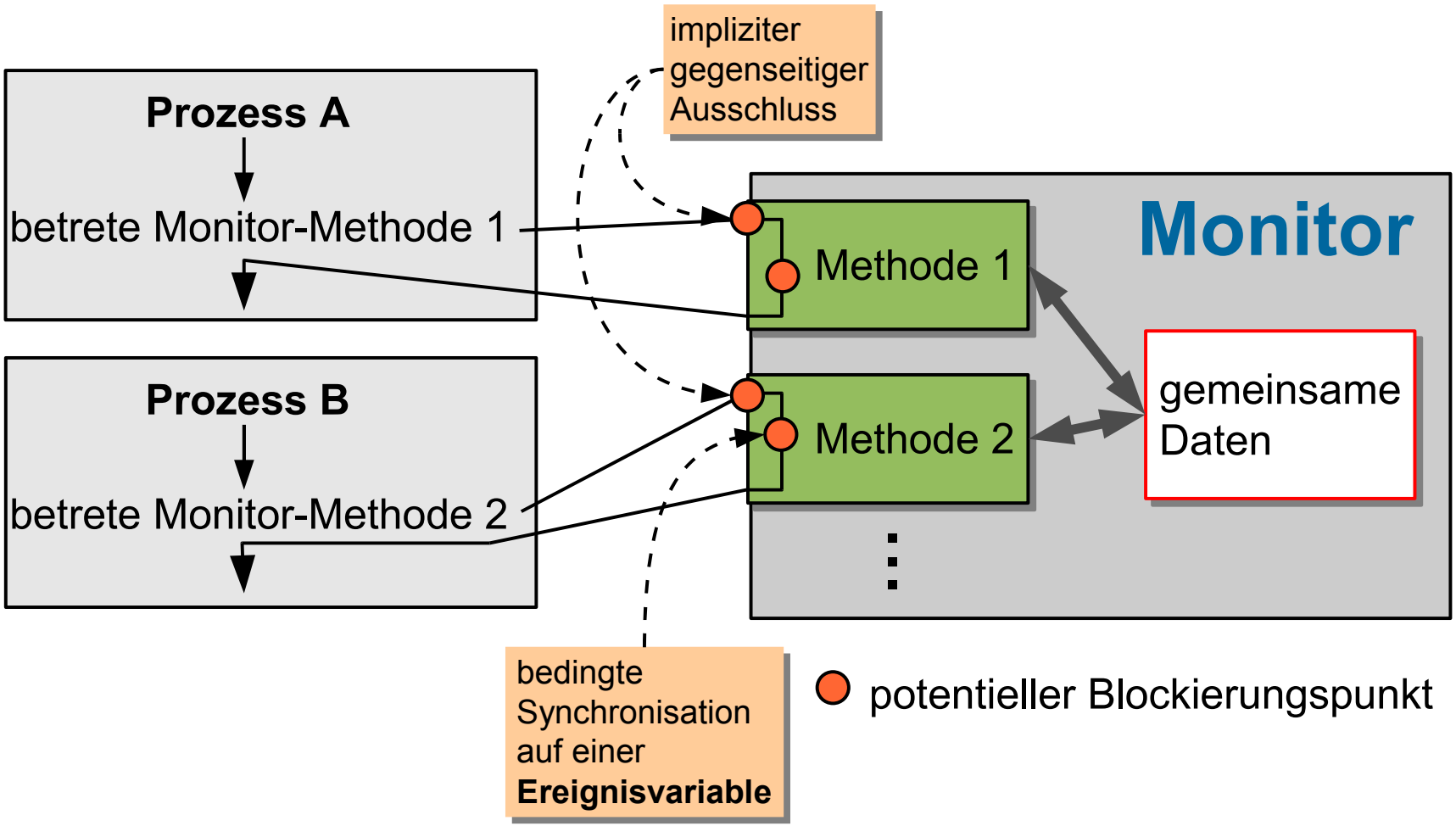
- Erweiterungen
  - nicht-blockierendes p()
  - *Timeout*
  - Felder von Zählern
- Fehlerquellen
  - Semaphorbenutzung wird nicht erzwungen
  - Abhängigkeit kooperierender Prozesse
    - jeder muss die Protokolle exakt einhalten
  - Aufwand bei der Implementierung

➔ Unterstützung durch die Programmiersprache

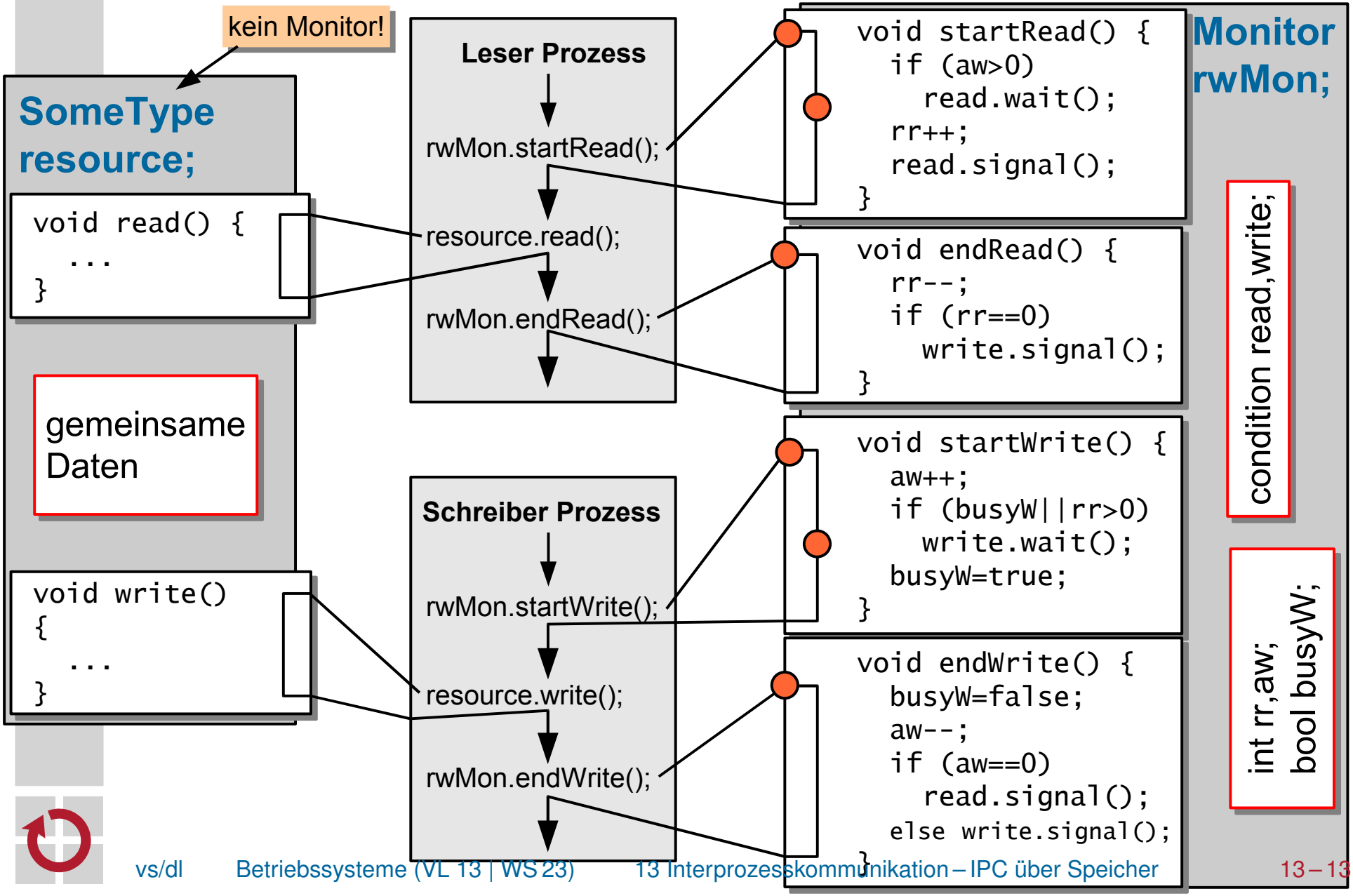
- Korrekte Synchronisation wird erzwungen

Monitore – synchronisierte ADTs [1]

- Ansatz: Abstrakte Datentypen werden mit Synchronisationseigenschaften gekoppelt



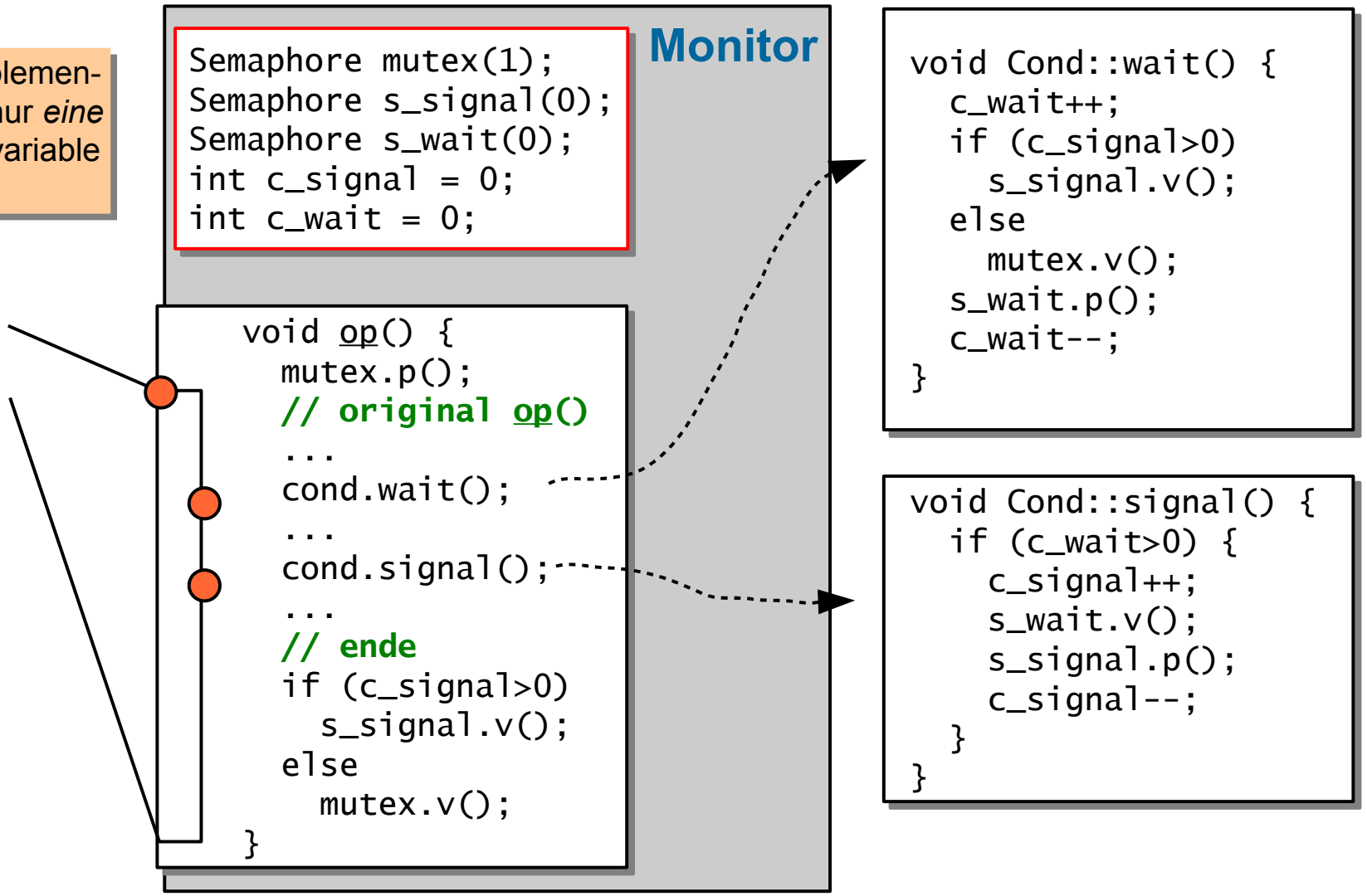
Monitore – Leser/Schreiber-Problem



Monitore – Implementierung

- ... auf Basis von Semaphoren

einfache Implementierung, die nur *eine* Bedingungsvariable unterstützt.



Monitore – Diskussion

- Einschränkung der Nebenläufigkeit auf vollständigen gegenseitigen Ausschluss.
  - in Java daher 'synchronized' auch für einzelne Methoden
- Kopplung von logischer Struktur und Synchronisation ist jedoch nicht immer natürlich.
  - siehe Leser/Schreiber Beispiel
  - gleiches Problem wie beim Semaphor: Programmierer müssen ein Protokoll einhalten

➔ Die Synchronisation sollte von der Organisation der Daten und Methoden besser getrennt werden.

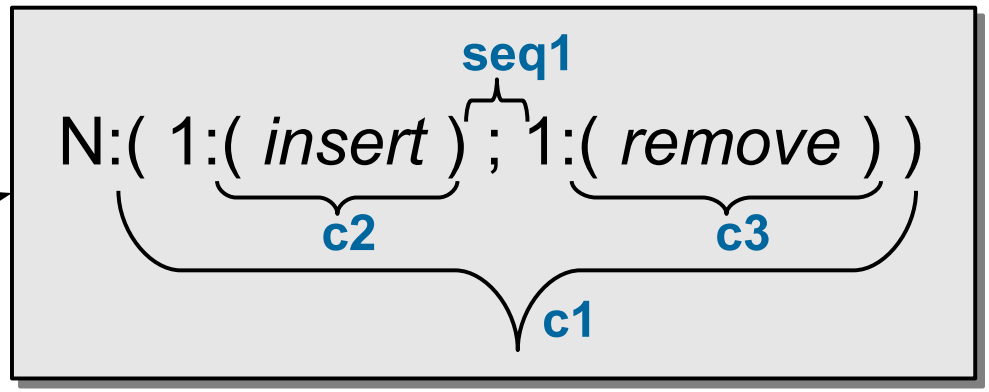
Pfadausdrücke [2]

- Idee: flexible Ausdrücke beschreiben erlaubte Reihenfolgen und den Grad der Nebenläufigkeit.
- **path** *name1*, *name2*, *name3* **end**
  - bel. Reihenfolge und bel. nebenläufige Ausführung von *name1-3*
- **path** *name1*; *name2* **end**
  - vor jeder Ausführung von *name2* mindestens einmal *name1*
- **path** *name1* + *name2* **end**
  - alternative Ausführung: entweder *name1* oder *name2*
- **path** 2:(*Pfadausdruck*) **end**
  - max. 2 Kontrollflüsse dürfen gleichzeitig im *Pfadausdruck* sein
- **path** N:(1:(insert); 1:(remove)) **end**
  - z.B. Synchronisation eines N-elementigen Puffers
    - gegenseitiger Ausschluss während *insert* und *remove*
    - vor jedem *remove* muss mindestens ein *insert* erfolgt sein
    - nie mehr als N abgeschlossene *insert*-Operationen

Pfadausdrücke – Implementierung (1)

- Transformation in Zustandsautomaten
  - Zustandsänderung bei Ein-/Austritt in die/aus der Operation
- Beispiel:

für jedes '*X*:(..)' und ';' wird ein Zähler eingeführt.




```
int c1=0;
int c2=0;
int c3=0;
int
seq1=0;
```

```
bool mayInsert () {
  return c1<N && c2<1;
}

void startInsert () {
  c1++; c2++;
}

void endInsert () {
  c2--; seq1++;
}
```

```
bool mayRemove () {
  return c1<N && seq1>0 && c3<1;
}

void startRemove () {
  c3++; seq1--;
}

void endRemove () {
  c3--; c1--;
}
```

Pfadausdrücke – Implementierung (2)

- Transformation der Operationen

für jede Operation wird ein Semaphor und ein Zähler eingeführt.

N:(1:(*insert*) ; 1:(*remove*))

sem1/csem1 sem2/csem2

```
Semaphore mutex(1);
int csem1=0;
Semaphore sem1(0);
int csem2=0;
Semaphore sem2(0);
```

```
void Insert() {
  mutex.p();
  if (!mayInsert()) {
    csem1++;
    mutex.v();
    sem1.wait();
  }
  startInsert();
  mutex.v();
  // original insert-Code
  mutex.p();
  endInsert();
  if (!wakeup())
    mutex.v();
}
```

```
bool wakeup() {
  if (csem1>0 &&
    mayInsert()) {
    csem1--;
    sem1.v();
    return true;
  }
  if (csem2>0 &&
    mayRemove()) {
    csem2--;
    sem2.v();
    return true;
  }
  return false;
}
```

Pfadausdrücke – Diskussion

- Vorteile
  - komplexere Interaktionsmuster als mit Monitoren möglich
    - read + 1: write
  - Einhaltung der Interaktionsprotokolle wird erzwungen
    - weniger Fehler!
- Nachteile
  - Synchronisationsverhalten kann nicht von Zustandsvariablen oder Parametern abhängen
    - Erweiterung: Pfadausdrücke mit Prädikaten
  - Synchronisation des Zustandsautomaten kann Flaschenhals werden
  - keine Unterstützung für Pfadausdrücke in gebräuchlichen Programmiersprachen

Thread-Safety-Analyse (clang) – Motivation

```
typedef struct { ... } lock_t;
extern void lock(lock_t *l);
extern void unlock(lock_t *l);

lock_t l;
int sum;

extern void alert(void);

int update(int c) {
        lock(&l);
        sum += c;
        if (0 <= sum) {
                unlock(&l);
                return 1;
        }
        alert();

        unlock(&l);
        return 0;
}
```

<= kann man vergessen

<= kann man leicht vergessen

<= kann man auch ohne lock aufrufen

<= kann man vergessen

... und der Compiler warnt nicht!

Thread-Safety-Analyse – Idee

Annotieren des Programms mit Attributen:

```
#define CAPABILITY(x)    attribute((capability(x)))
#define GUARDED_BY(x)    attribute((guarded_by(x)))
#define ACQUIRE(x)       attribute((acquire_capability(x)))
#define RELEASE(x)       attribute((release_capability(x)))
#define REQUIRES(x)      attribute((requires_capability(x)))
```

- das Halten eines Locks ist eine „Capability” (Befähigung)
- Zugriffe auf Variablen können geschützt sein („guarded_by”)
- Capabilities kann man erlangen („acquire”)
- Capabilities kann man abgeben („release”)
- Capabilities können notwendig sein („require”)

Thread-Safety-Analyse – Umsetzung

```
typedef struct { ... } lock_t CAPABILITY;
extern void lock(lock_t *l) ACQUIRE(l);
extern void unlock(lock_t *l) RELEASE(l);

lock_t l;
int sum GUARDED_BY(l);

extern void alert(void) REQUIRES(l);

int update(int c) {
        lock(&l);
        sum += c;
        if (0 <= sum) {
                unlock(&l);
                return 1;
        }
        alert();

        unlock(&l);
        return 0;
}
```

... und der Compiler warnt ggf.!

Thread-Safety-Analyse – virtuelle Capabilities

```
typedef struct { ... } irq_disabled_t CAPABILITY;
extern irq_disabled_t irq_disabled;
extern void irq_disable(void) ACQUIRE(irq_disabled);
extern void irq_enable(void) RELEASE(irq_disabled);

char buf GUARDED_BY(irq_disabled);

char kbd_get(void)
{
        irq_disable();
        char c = buf;
        irq_enable();
        return c;
}

int kbd_prologue(void) REQUIRES(irq_disabled)
{
        buf = ...;
        return 1;
}
```

Aber: `kbd_get` könnte mit ausgeschalteten Interrupts aufgerufen werden!

Thread-Safety-Analyse – virtuelle Capabilities

```
typedef struct { ... } irq_enabled_t CAPABILITY;
typedef struct { ... } irq_disabled_t CAPABILITY;
extern irq_enabled_t irq_enabled;
extern irq_disabled_t irq_disabled;
extern void irq_disable(void) RELEASE(irq_enabled) ACQUIRE(irq_disabled
extern void irq_enable(void) RELEASE(irq_disabled) ACQUIRE(irq_enabled)

char buf GUARDED_BY(irq_disabled);

char kbd_get(void) REQUIRES(irq_enabled)
{
        irq_disable();
        char c = buf;
        irq_enable();
        return c;
}

int kbd_prologue(void) REQUIRES(irq_disabled)
{
        buf = ...;
        return 1;
}
```

Deadlocks – Wiederholung

Vier Bedingungen für einen Deadlock (mit Locks):

1. Locks können nicht entzogen werden
2. Fäden halten Locks und wollen weitere
3. Locks können nicht geshared werden
4. es können Zyklen auftreten

Deadlock-Verhinderung:

- Punkte 1-3 können i.A. nicht geändert werden
- Punkt 4: Zyklen verbieten

=> Lock-Ebenen einführen

Thread-Safety-Analyse – Deadlock-Verhinderung

```
extern void lock_0(lock_0_t *l) RELEASE(level_0) ACQUIRE(l);
extern void unlock_0(lock_0_t *l) RELEASE(l) ACQUIRE(level_0);
extern void lock_1(lock_1_t *l) RELEASE(level_1) ACQUIRE(l);
extern void unlock_1(lock_1_t *l) RELEASE(l) ACQUIRE(level_1);

lock_0_t l_0;
int var_0 GUARDED_BY(l_0);
lock_1_t l_1;
int var_1 GUARDED_BY(l_1);

void bar(void) REQUIRES(level_0)
{
        lock_0(&l_0);
        var_0 = ...;
        unlock_0(&l_0);
}

void foo(void) REQUIRES(level_0) REQUIRES(level_1)
{
        lock_1(&l_1);
        var_1 = ...;
        bar();
        unlock_1(&l_1);
}
```

Thread-Safety-Analyse (clang)

Zusammenfassung/Erfahrungen clang-Thread-Safety-Analyse:

- elementare Funktionen, die Capabilities erzeugen/abgeben, müssen mit „NO_THREAD_SAFETY_ANALYSIS” markiert werden

```
void irq_disable(void)
        RELEASE(irq_enabled) ACQUIRE(irq_disabled)
        NO_THREAD_SAFETY_ANALYSIS
{
        asm volatile ("cli");
}
```

- Funktionszeiger können nicht attributiert werden
- ist eine Art boolsche „und”-Verknüpfung
- „nicht”-Operator über inverse Capabilities
- Tipp-Arbeit
- dokumentiert Code

JITTY Erfahrungen: **sehr** wertvoll!

Agenda

Einordnung
IPC über Speicher
IPC über Nachrichten
Basisabstraktionen
Trennung der Belange mit AOP
Zusammenfassung

IPC über Nachrichten

- Anwendungsfälle/Voraussetzungen
  - IPC über Rechnergrenzen
  - Interaktion isolierter Prozesse
- positive Eigenschaften:
  - einheitliches Paradigma für IPC mit lokalen und entfernten Prozessen
  - ggf. Pufferung und Synchronisation
  - Indirektion erlaubt transparente Protokollerweiterungen
    - Verschlüsselung, Fehlerkorrektur, ...
  - Hochsprachenmechanismen wie OO-Nachrichten oder Prozeduraufrufe lassen sich gut auf IPC über Nachrichten abbilden (RPC, RMI)

Nachrichtenbasierte Kommunikation

- Bekannt (aus SOS):
  Variationen von `send()` und `receive()`
  - synchron/asynchron (blockierend/nicht blockierend)
  - gepuffert/ungepuffert
  - direkt/indirekt
  - feste Nachrichtengröße/variable Größe
  - symmetrische/asymmetrische Kommunikation
  - mit/ohne *Timeout*
  - *Broadcast/Multicast*

Agenda

Einordnung
IPC über Speicher
IPC über Nachrichten
Basisabstraktionen
- Windows/Unix/...
- Dualität der Konzepte

Trennung der Belange mit AOP
Zusammenfassung

Basisabstraktionen

- Welche IPC Basisabstraktionen bieten Betriebssysteme?
  - UNIX-Systeme: Sockets, System V Semaphore, Messages, Shared Memory
  - Windows NT/2000/XP: Shared Memory, Events, Semaphore, Mutant (Mutex), Sockets, Pipes, Named Pipes, Mailslots, ...
  - Mach: Nachrichten an Ports und Shared Memory (mit *Copy on Write*)
- Welche Abstraktionen nutzen die Systeme i.d.R. intern?
  - Semaphore erlauben gegenseitigen Ausschluss und einseitige Synchronisation, also sehr häufige Anwendungsfälle
    - werden praktisch immer benutzt
  - Mikrokerne und verteilte Betriebssysteme: Nachrichten
  - Monolithische Systeme: Semaphore und gemeinsamen Speicher

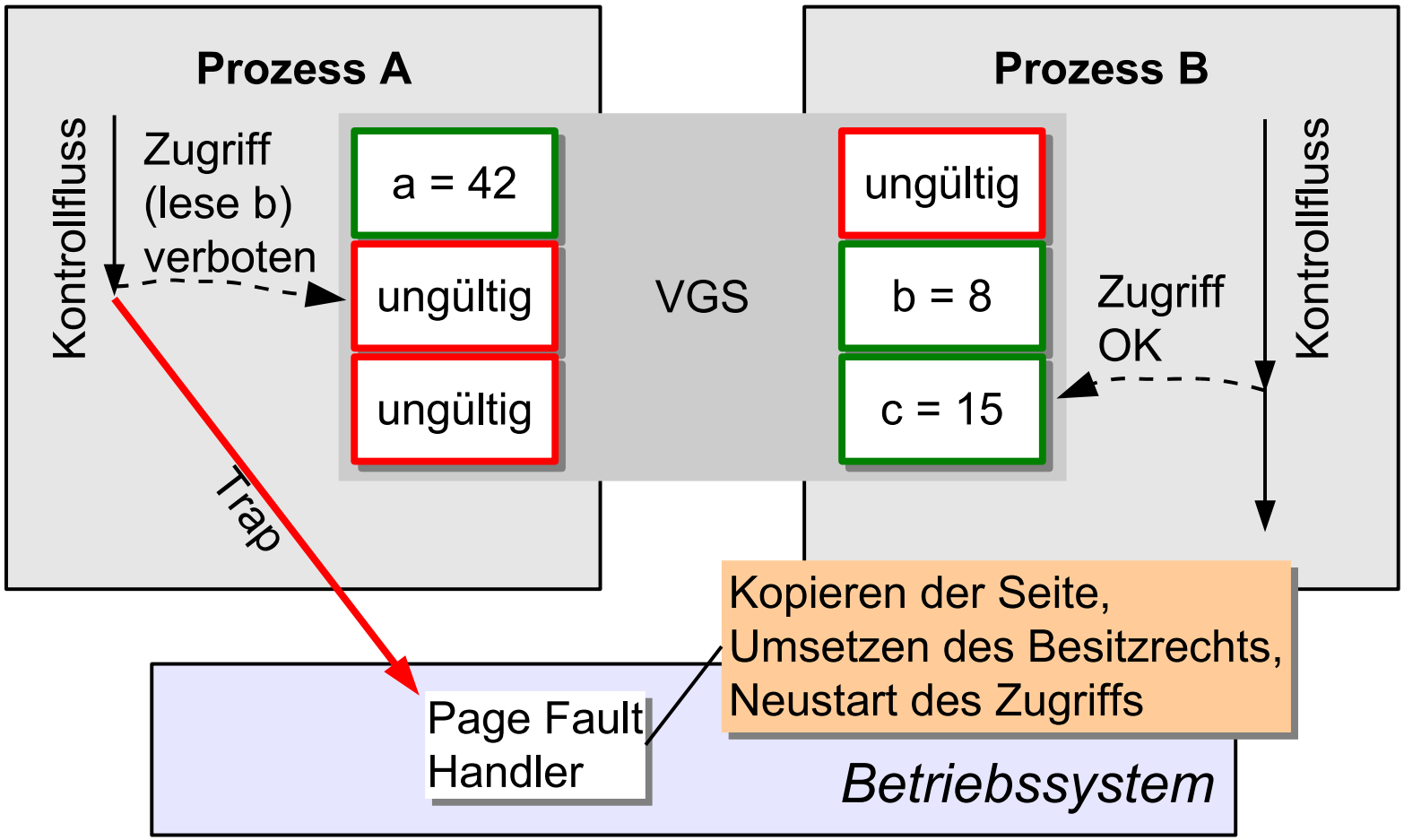
Dualität – Nachrichten in gemeinsamem Speicher

- auf Basis von Semaphoren und gemeinsamem Speicher lässt sich leicht eine *Mailbox*-Abstraktion realisieren:
  - Nachrichten werden nicht kopiert
    - Sender sorgt für Speicher
  - `receive` blockiert ggf.
  - Mailbox-Abstraktion erlaubt M:N IPC

```
class Mailbox : public List {
  Semaphore mutex;    // (1)
  Semaphore has_elem; // (0)
public:
  Mailbox() : mutex(1), has_elem(0) {}
  void send(Message *msg) {
    mutex.p();
    enqueue(msg); // aus List
    mutex.v();
    has_elem.v();
  }
  Message *receive() {
    has_elem.p();
    mutex.p();
    Message *result = dequeue (); // List
    mutex.v();
    return result;
  }
};
```

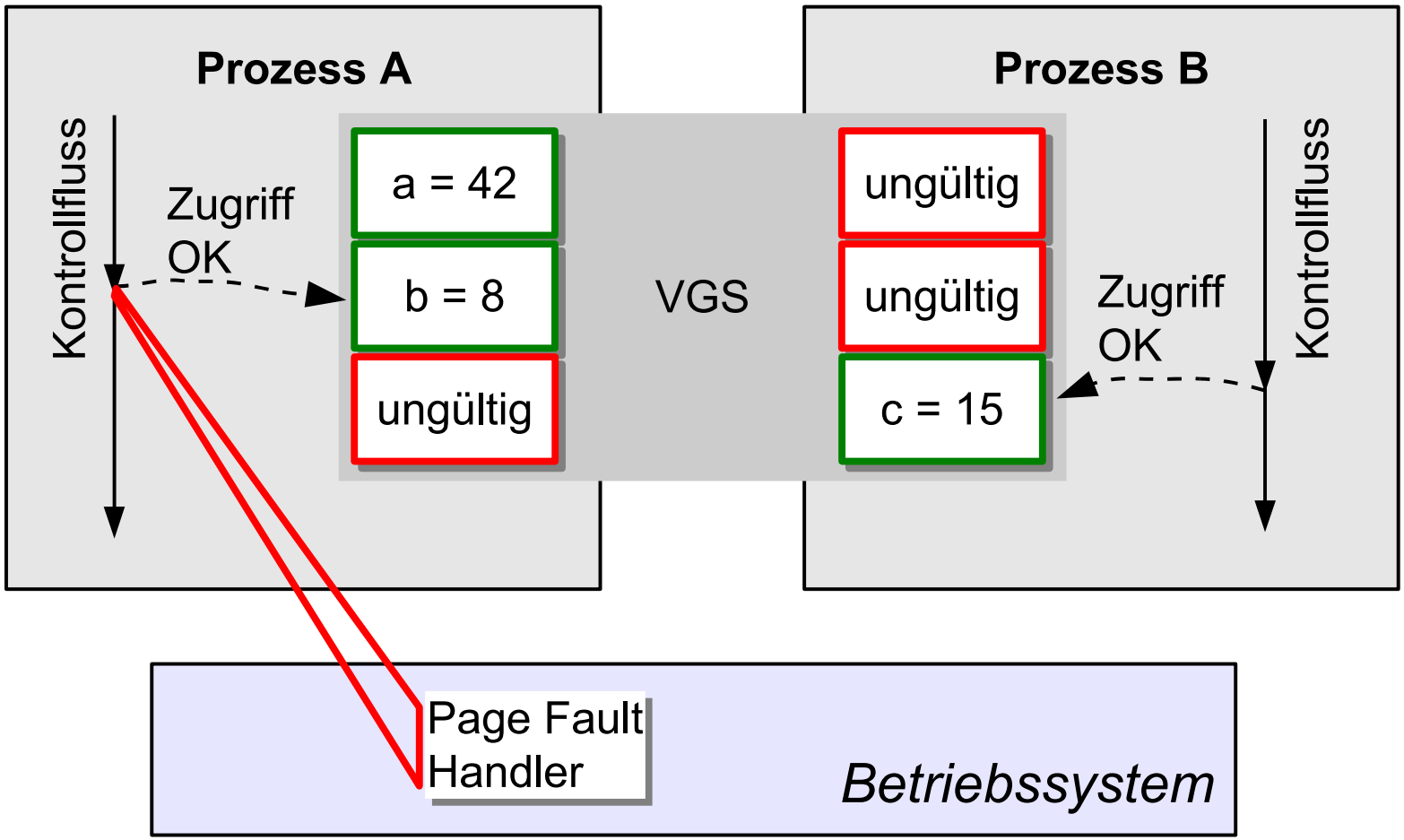
Dualität – Gemeinsamer Speicher mit Nachrichten

- „Virtueller gemeinsamer Speicher“ (VGS [3])



Dualität – Gemeinsamer Speicher mit Nachrichten

- „Virtueller gemeinsamer Speicher“ (VGS)



Dualität – VGS Diskussion

- Verteilter virtueller gemeinsamer Speicher ermöglicht...
  - das Programmiermodell von Multiprozessoren auf Mehrrechnersystemen zu nutzen
  - IPC über (virtuellen) gemeinsamen Speicher trotz getrennter Adressräume
- Probleme:
  - Latenzen der Kommunikation und Trap-Behandlung
  - „*false sharing*“ - Seitengröße entspricht nicht Objektgröße
- Lösungsansätze:
  - schwache Konsistenzmodelle, z.B.:
    - nicht jeder Zugriff führt zu einem Trap, veraltete Werte werden in Kauf genommen
  - Änderungen asynchron per *Broad-/Multicast* verbreiten

Dualität – Aktive Objekte

- Objekte mit Kontrollfluss
- gut geeignet zur Zugriffssynchronisation in Systemen mit nachrichtenbasierter IPC

```
void client1() {
  Message msg(DO_THIS);
  send(srv, msg);
}
```

```
void client2() {
  Message msg(DO_THAT);
  send(srv, msg);
}
```

Gegenseitiger Ausschluss durch die Verarbeitungsschleife wird garantiert. Durch das synchrone `send()` blockiert ein Client solange der Server noch beschäftigt ist.

→ genau wie ein **Monitor**

```
class Server : public ActiveObject {
  Msg msg; // Nachrichtenpuffer
public:
  ...
  // Objekt mit Kontrollfluss!
  void action() {
    while (true) {
      receive(ANY, msg); // empfange Nachr.
      switch (msg.type()) {
        case DO_THIS:  doThis(); break;
        case DO_THAT:  doThat(); break;
        default:       handleError();
      }
      reply(msg);
    }
  }
};
```

Dualität – Aktive Objekte

- Leser/Schreiber Problem mit Nachrichtenaustausch

```
void reader() {
  Msg start_read(START_READ);
  send(srv, start_read);
  Msg read_msg(DO_READ);
  send(srv, read_msg);
  Msg end_read(END_READ);
  send(srv, end_read);
  // benutze Daten in 'read_msg'
}
```

```
void writer() {
  Msg start_write(START_WRITE);
  send(srv, start_write);
  // hier Nachricht füllen
  Msg write_msg(DO_WRITE);
  send(srv, write_msg);
  Msg end_write(END_WRITE);
  send(srv, end_write);
}
```

```
class RWServer : public ActiveObject {
  Msg msg; // Nachrichtenpuffer
public:
  ...
  // Kontrollfluss
  void action() {
    while (true) {
      receive(ANY, msg); // empfange N.
      switch (msg.type()) {
      case START_READ:  startRead();  break;
      case DO_READ:     doRead();     break;
      case END_READ:    endRead();    break;
      case START_WRITE: startWrite(); break;
      case DO_WRITE:    doWrite();    break;
      case END_WRITE:   endWrite();   break;
      default: msg.type(ERROR); reply(msg);
      }
    }
  }
};
```

Dualität – Aktive Objekte

- Leser/Schreiber Problem mit Nachrichtenaustausch
  - die eigentliche Lese- und Schreiboperation erfolgt nebenläufig durch einen Kindprozess

die 'request' Nachricht muss kopiert werden, da sie während der Ausführung des Kindprozesses überschrieben werden könnte

```
void RWServer::doRead() {
  Msg copy=msg;
  if (fork()==0) {
    // das eigentliche Lesen
    copy.set(...) // Antwort
    reply(copy);
  }
  else {
  } // Elternprozess: nichts
}
```

```
void RWServer::doWrite() {
  Msg copy=msg;
  if (fork()==0) {
    // das eigentlich
Schreiben
    // (benutzt 'copy')
    reply(copy);
  }
  else {
  } // Elternprozess: nichts
}
```

der Server-Prozess kann sofort wieder auf 'requests' warten

Dualität – Aktive Objekte

- Leser/Schreiber Problem mit Nachrichtenaustausch

```
void RWServer::startRead() {
  ar++;
  if (aw>0)
    read.copy_enqueue(msg);
  else {
    rr++; reply(msg);
  }
}

void RWServer::endRead() {
  ar--; rr--;
  if (rr==0 && aw>0) {
    Msg wmsg=write.dequeue();
    ww++; reply(wmsg);
  }
  reply(msg);
}
```

```
void RWServer::startWrite() {
  aw++;
  if (ww>0 || rr>0)
    write.copy_enqueue(msg);
  else {
    ww++; reply(msg);
  }
}

void RWServer::endWrite() {
  aw--; ww--;
  if (aw>0) {
    Msg wmsg=write.dequeue();
    ww++; reply(wmsg);
  }
  else while (rr < ar) {
    Msg rmsg=read.dequeue();
    rr++; reply(rmsg);
  }
  reply(msg);
}
```

Ergebnis: Die Semantik / Parallelität entspricht der Monitor-basierten Implementierung.

Dualität – Diskussion

- Gibt es einen fundamentalen Unterschied zwischen IPC über gem. Speicher und IPC über Nachrichten?
  - zugespitzt: sind oder prozedurorientierte BS (Monolithen) oder prozessorientierte BS (Mikrokerne) besser?
- Beispiel: Leser/Schreiber Monitor vs. *Server*:
  - Monitor: 2 potentielle Wartepunkte
    - Client wird verzögert für gegenseitigen Ausschluss.
    - Client wird ggf. wegen einer Ereignisvariablen weiter verzögert.
  - *Server*: 2 potentielle Wartepunkte
    - *Reply* wird verzögert, da der *Server* noch andere *Requests* bearbeitet.
    - *Reply* wird ggf. weiter verzögert, wenn der *Request* in eine Warteschlange gehängt werden muss.
- Fazit: Dualität in Synchronisation und Nebenläufigkeit [4]

Agenda

Einordnung
IPC über Speicher
IPC über Nachrichten
Basisabstraktionen
Trennung der Belange mit AOP
Zusammenfassung

Trennung der Belange mittels AOP

- „Aspektorientierte Programmierung“ erlaubt die *modulare* Implementierung „querschneidender“ Belange
- Beispiel in AspectC++:

```
// Festlegung der Monitore des Systems
pointcut monitors() = "FileTable"||"BufferCache";

// Synchronisation per Aspekt
aspect MonitorSynch {
  advice monitors() : slice struct {
    Semaphore _mutex;
  };
  advice construction(monitors()) : before() {
    tjp->that()->_mutex.init(1);
  }
  advice execution(monitors()) : around() {
    tjp->that()->_mutex.p(); // Monitor sperren
    tjp->proceed();          // Fkt. ausführen
    tjp->that()->_mutex.v(); // Monitor freigeben
  }
};
```

"Einfügung" eines Semaphors in die Monitor-Klassen

"Code-Advice" für Ereignisse im Programmablauf

Agenda

Einordnung
IPC über Speicher
IPC über Nachrichten
Basisabstraktionen
Trennung der Belange mit AOP
Zusammenfassung

Zusammenfassung und Ausblick

- Es gibt zwei Hauptklassen von IPC Mechanismen:
  - IPC über gemeinsamen Speicher
  - nachrichtenbasierte IPC
- Mechanismen beider Klassen sind in realen Betriebssystemen anzutreffen
  - Sprachmechanismen wie Monitore und Pfadausdrücke können bei der BS-Entwicklung allerdings i.d.R. nicht verwendet werden
- Bzgl. des Synchronisationsverhaltens und dem Grad der Nebenläufigkeit zeichnet sich keine Klasse besonders aus
  - Vor- und Nachteile liegen woanders
  - Ausblick: mit AOP Techniken könnte man von den konkreten Kommunikations- und Synchronisations*mechanismen* abstrahieren

Literatur

[1] C. A. R. Hoare, Monitor – An Operating System Structuring Concept, Communications of the ACM 17, 10, S. 549-557, 1974

[2] R. H. Campbell and A. N. Habermann, The Specification of Process Synchronization by Path Expressions, Lecture Note in Computer Science 16, Springer, 1974

[3] K. Li, Shared Virtual Memory on Loosely Coupled Multiprocessors, PhD Thesis, Yale University, 1986

[4] Lauer, H. C. and Needham, R. M. 1979. On the duality of operating system structures. SIGOPS Oper. Syst. Rev. 13, 2 (Apr. 1979), 3-19

Betriebssysteme (BS)

VL 14 – Zusammenfassung und Ausblick

Volkmar Sieh / **Daniel Lohmann**

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 7. Februar 2024

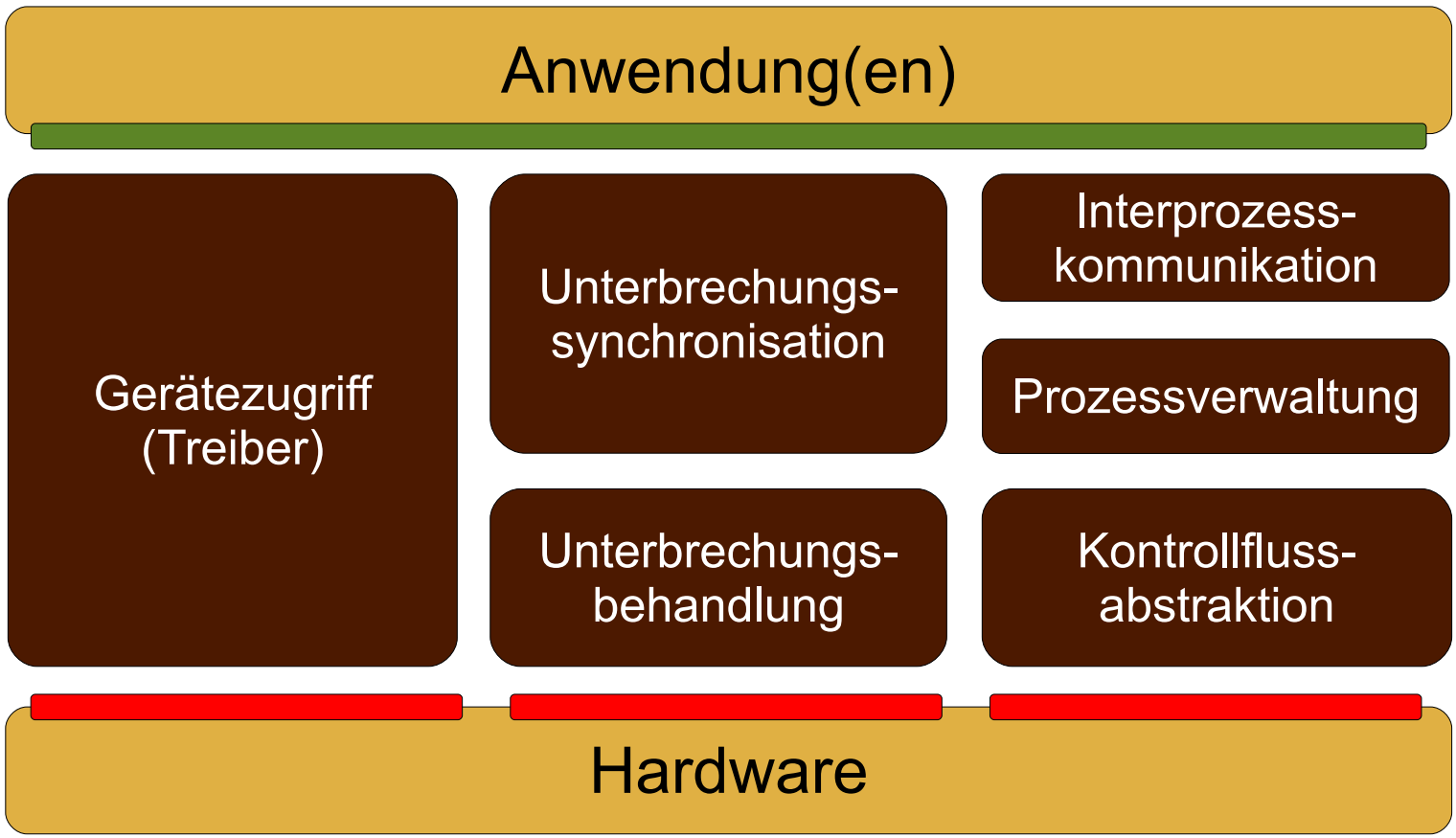
https://sys.cs.fau.de/lehre/ws23/bs

Lernziele

⇝VL 1

- **Vertiefen** des Wissens über die interne Funktionsweise von Betriebssystemen
  - Ausgangspunkt: Systemprogrammierung
  - Schwerpunkt: Nebenläufigkeit und Synchronisation
- **Entwickeln** eines Betriebssystems *von der Pike auf*
  - OOStuBS / MPStuBS Lehrbetriebssysteme
  - Praktische Erfahrungen im Betriebsystembau machen
- **Verstehen** der technologischen Hardware-Grundlagen
  - PC-Technologie verstehen und einschätzen können
  - Schwerpunkt: Intel x86_64

Überblick Vorlesungen



Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

- VL_1 ***Einführung***
- VL_2 ***BS-Entwicklung***
- VL_3 ***IRQs (Hardware)***
- VL_4 ***IRQs (Software)***
- VL_5 ***IRQs (SoftIRQ)***
- VL_6 ***IRQs (Synchronisation)***
- VL_7 ***Intel IA-32***
- VL_8 ***Koroutinen und Fäden***
- VL_9 ***Scheduling***
- VL_{10} ***Architekturen***
- VL_{11} ***Fadensynchronisation***
- VL_{12} ***Gerätetreiber***
- VL_{13} ***IPC***

Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

1. Ein Streifzug durch die PC-Architektur

- VL_1 ***Einführung***
- VL_2 ***BS-Entwicklung***
- **VL_3** ***IRQs (Hardware)***
- VL_4 ***IRQs (Software)***
- VL_5 ***IRQs (SoftIRQ)***
- VL_6 ***IRQs (Synchronisation)***
- **VL_7** ***Intel IA-32***
- VL_8 ***Koroutinen und Fäden***
- VL_9 ***Scheduling***
- VL_{10} ***Architekturen***
- VL_{11} ***Fadensynchronisation***
- VL_{12} ***Gerätetreiber***
- VL_{13} ***IPC***

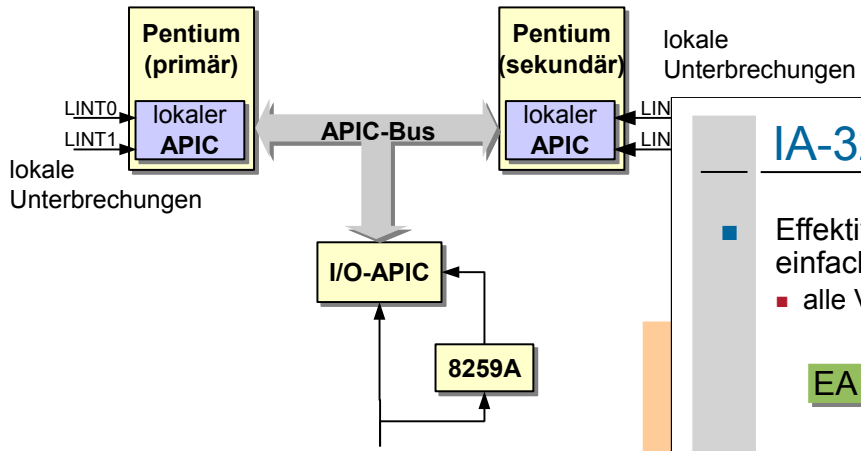
Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

1. Ein Streifzug durch die PC-Architektur

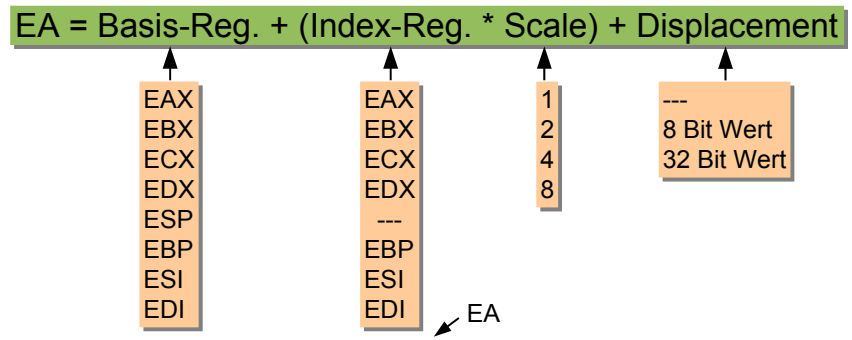
Die APIC Architektur

- ein APIC *Interrupt*-System besteht aus lokalen APICs auf jeder CPU und einem I/O APIC



IA-32: Adressierungsarten

- Effektive Adressen (EA) werden nach einem einfachen Schema gebildet
  - alle Vielzweckregister können dabei gleichwertig verwendet werden



- Beispiel: MOV EAX, **Feld[ESI * 4]**
  - Lesen aus Feld mit 4 Byte großen Elementen und ESI als Index

Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

2. Kontrollflüsse und ihre Interaktionen

- VL_1 ***Einführung***
- VL_2 ***BS-Entwicklung***
- **VL_3** ***IRQs (Hardware)***
- **VL_4** ***IRQs (Software)***
- **VL_5** ***IRQs (SoftIRQ)***
- **VL_6** ***IRQs (Synchronisation)***
- VL_7 ***Intel IA-32***
- **VL_8** ***Koroutinen und Fäden***
- **VL_9** ***Scheduling***
- VL_{10} ***Architekturen***
- **VL_{11}** ***Fadensynchronisation***
- VL_{12} ***Gerätetreiber***
- **VL_{13}** ***IPC***

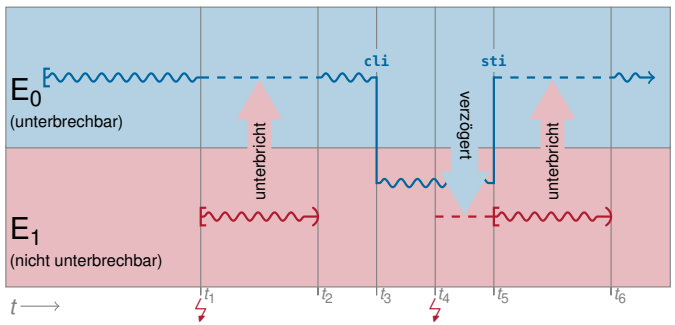
Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

2. Kontrollflüsse und ihre Interaktionen

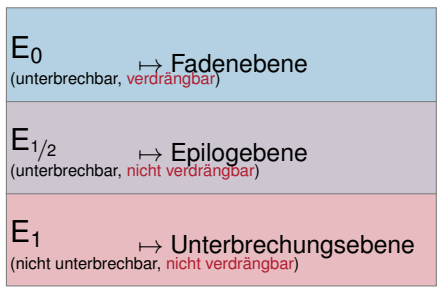
Prioritätsebenenmodell

- Kontrollflüsse können die Ebene wechseln
  - Mit `cli` wechselt ein E_0-Kontrollfluss explizit auf E_1
    - er ist ab dann nicht mehr unterbrechbar
    - andere E_1-Kontrollflüsse werden verzögert (↩ Sequentialisierung)
  - Mit `sti` wechselt ein E_1-Kontrollfluss explizit auf E_0
    - er ist ab dann (wieder) unterbrechbar
    - anhängige E_1-Kontrollflüsse „schlagen durch“ (↩ S



Erweitertes Prioritätsebenenmodell

- Kontrollflüsse auf E_l werden
  1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
  2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
  3. jederzeit verdrängt durch Kontrollflüsse von E_l (für $l = 0$)



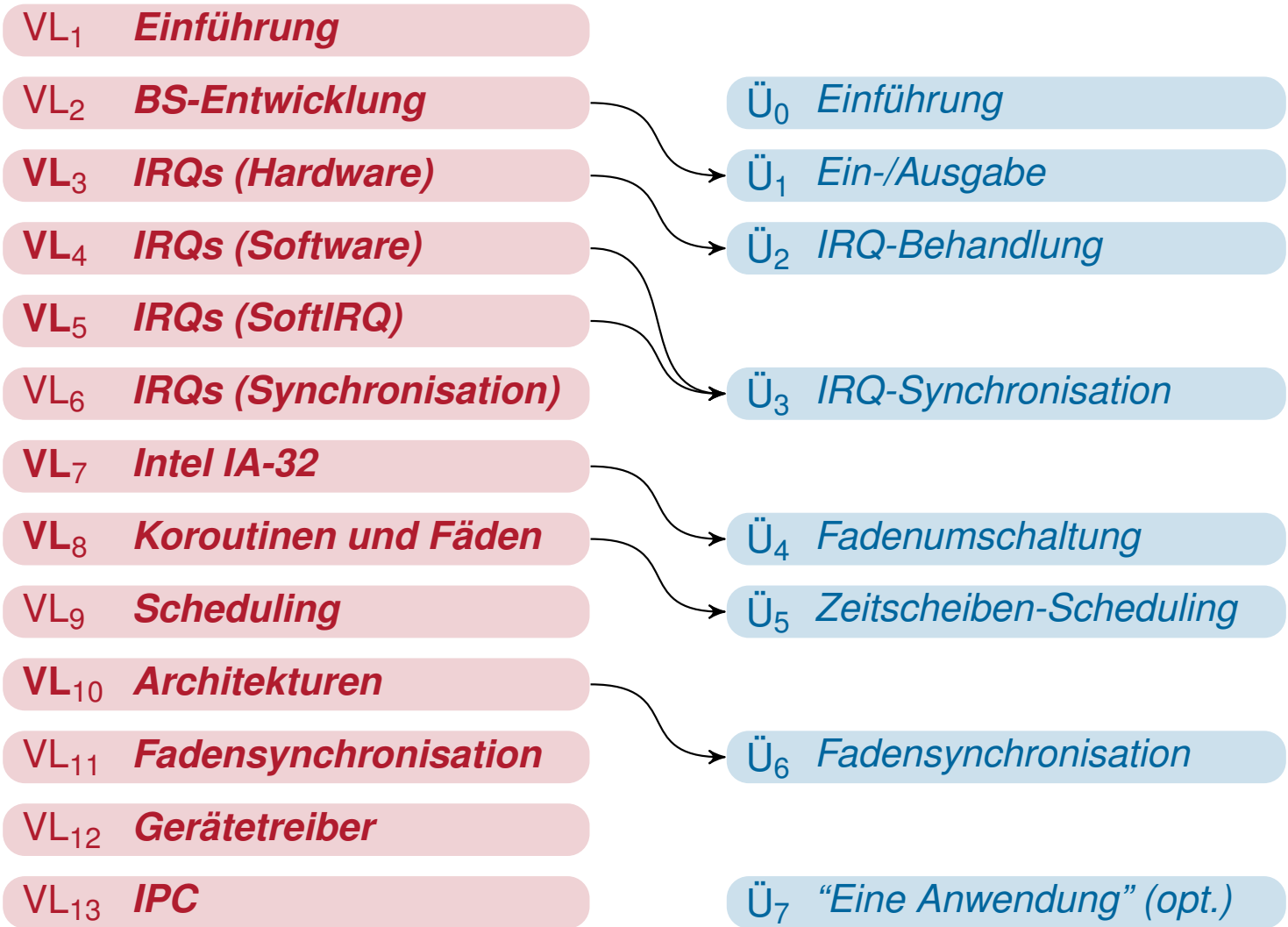
Kontrollflüsse der E_0 (Fadenebene) sind **verdrängbar**.

Für die Konsistenzssicherung auf dieser Ebene brauchen wir zusätzliche **Mechanismen** zur **Fadensynchronisation**.

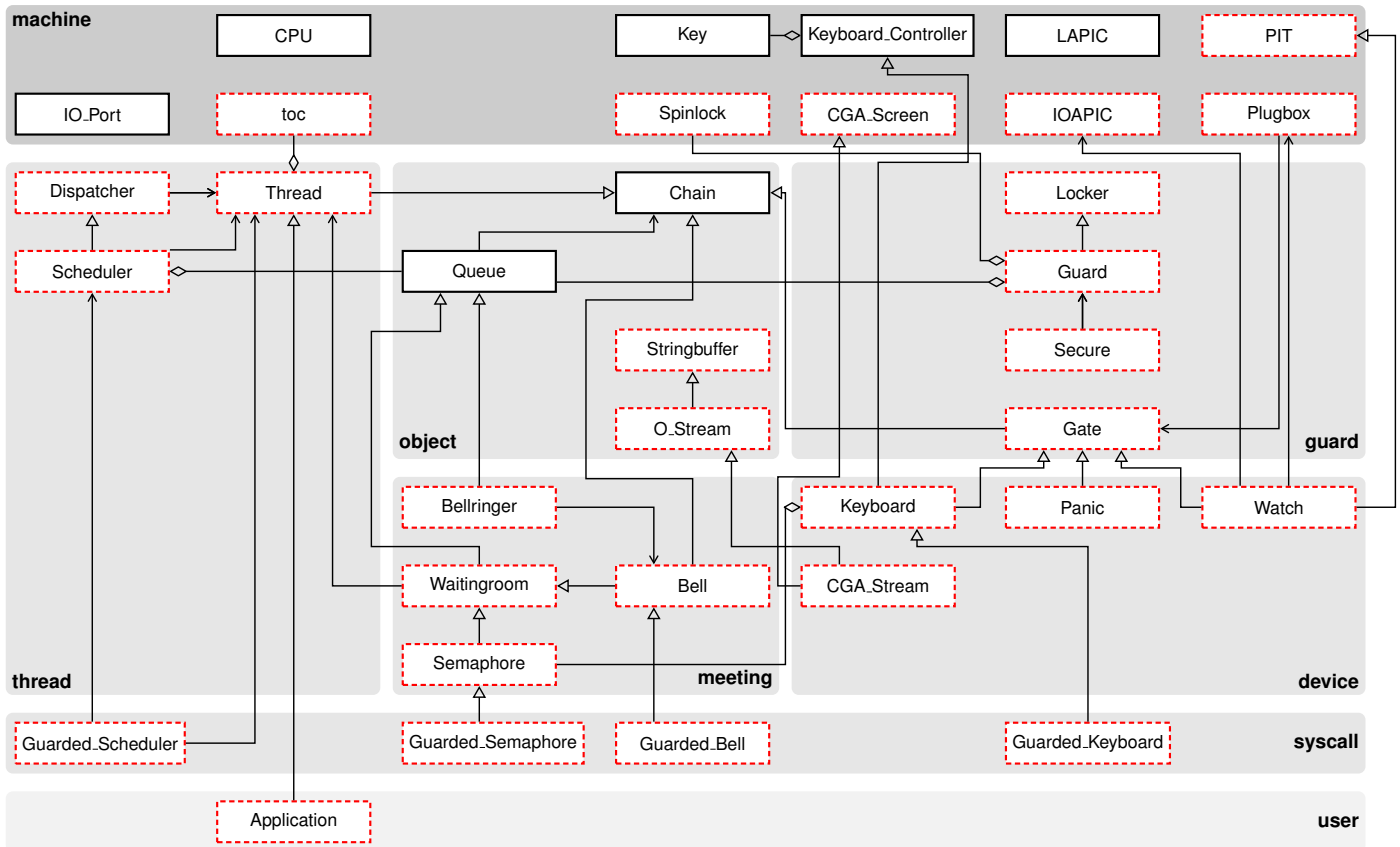
Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

2. Kontrollflüsse und ihre Interaktionen



2. Kontrollflüsse und ihre Interaktionen



Was wir gemacht haben

Drei inhaltliche Schwerpunkte!

3. BS-Konzept allgemein und am Beispiel (Windows/Linux)

- VL_1 ***Einführung***
- **VL_2** ***BS-Entwicklung***
- VL_3 ***IRQs (Hardware)***
- VL_4 ***IRQs (Software)***
- VL_5 ***IRQs (SoftIRQ)***
- VL_6 ***IRQs (Synchronisation)***
- VL_7 ***Intel IA-32***
- **VL_8** ***Koroutinen und Fäden***
- **VL_9** ***Scheduling***
- **VL_{10}** ***Architekturen***
- **VL_{11}** ***Fadensynchronisation***
- **VL_{12}** ***Gerätetreiber***
- **VL_{13}** ***IPC***

Was wir gemacht haben

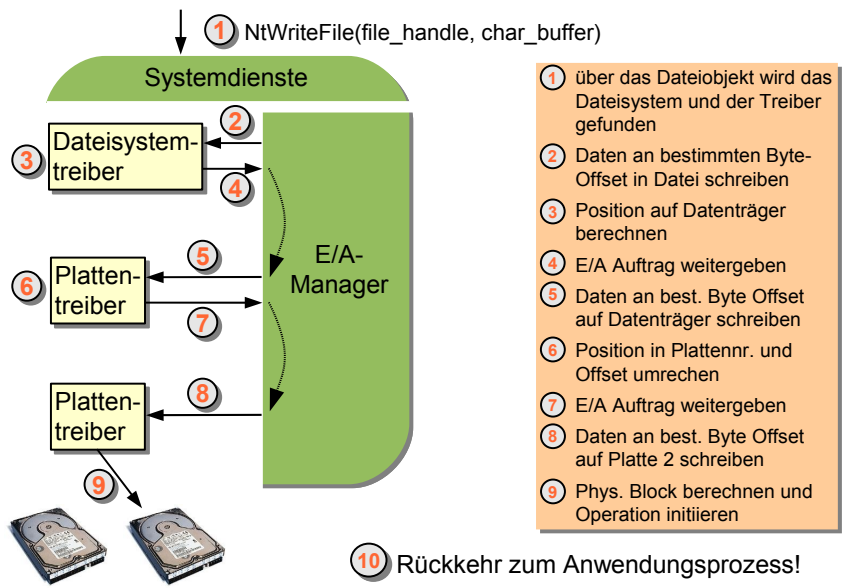
Drei inhaltliche Schwerpunkte!

3. BS-Konzept allgemein und am Beispiel (Windows/Linux)

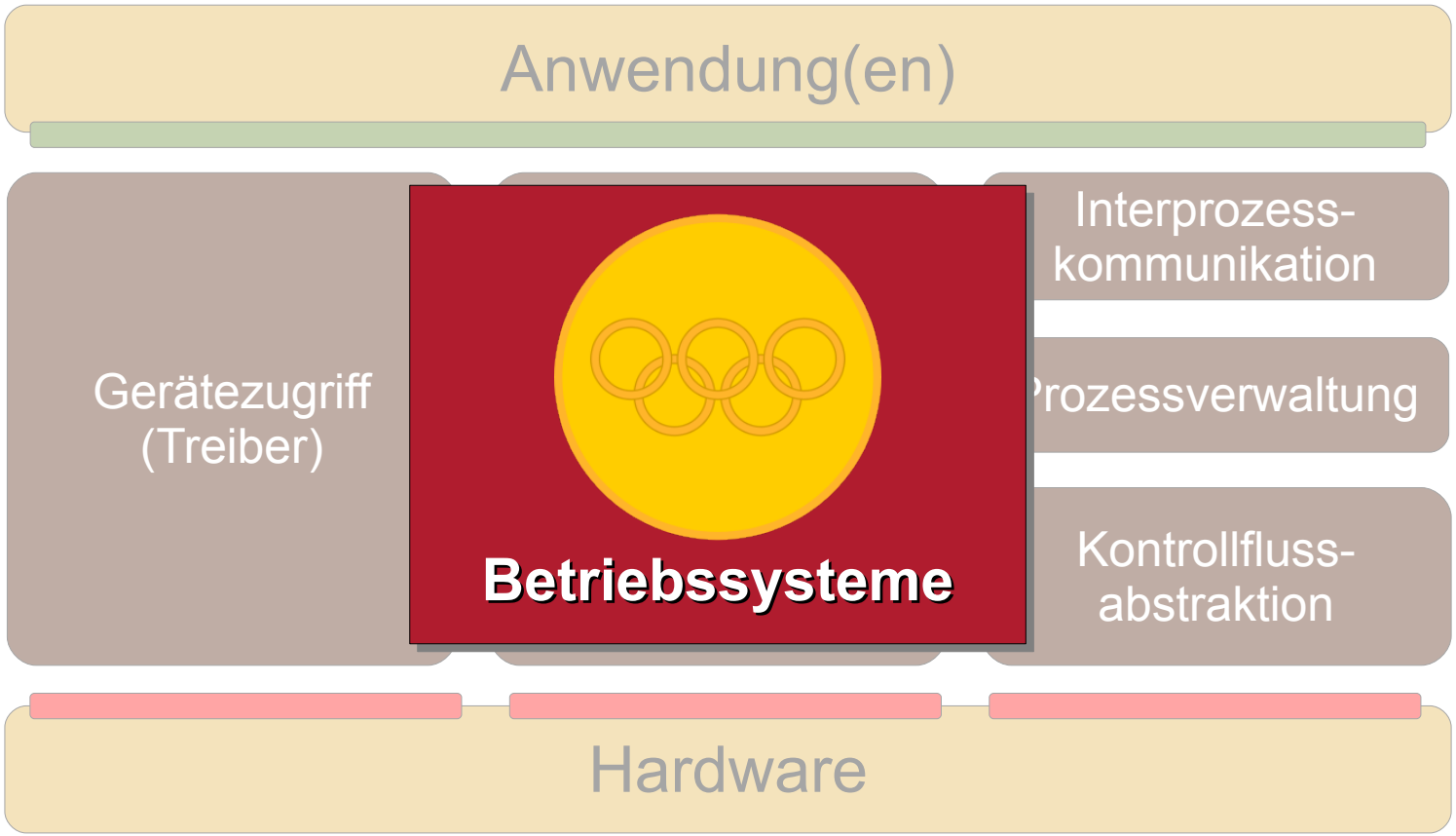
Completely Fair Scheduler (CFS)

- Ansatz: Ablaufbereite Tasks bekommen die Rechenzeit gleichmäßig ("fair") zugeteilt
  - bei *n* Tasks jeweils 1/*n*-tel der CPU-Leistung
  - hierarchische Zuteilung durch *scheduling groups*
- CFS läuft nur bei SCHED_NORMAL
  - Echtzeittask (SCHED_RR und SCHED_FIFO) w
  - ansonsten: Task mit *geringster* CPU-Zeit hat hö
- Scheduling-Kriterium ist die bislang zugeteilte
  - Ready-Liste als Rot-Schwarz-Baum, sortiert nac
  - Komplexität O(log *N*)
    (in der Praxis trotzdem effizienter als alter O(1)-S
  - Prioritäten (im Sinne von nice) werden durch "schnellere/langsamere" Uhren abgebildet

Windows – typischer E/A-Ablauf



Zusammen eine ganze Menge!



Realitätscheck: MPStuBS $\leftrightarrow$ „richtiges BS“

Es fehlt noch eine ganze Menge!

- Adressraumverwaltung und Prozesskonzept $\rightsquigarrow$ [BST]
- Dateisystem und Programmlader
- Netzwerk und TCP/IP
- ...

Realitätscheck: MPStuBS $\leftrightarrow$ „richtiges BS“

Es fehlt noch eine ganze Menge!

- Adressraumverwaltung und Prozesskonzept $\rightsquigarrow$ [BST]
- Dateisystem und Programmlader
- Netzwerk und TCP/IP
- ...

Beispiel Linux [14]

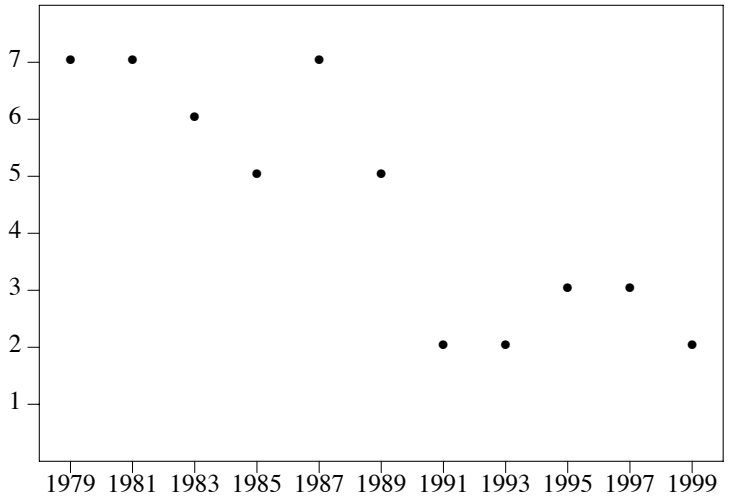
Aug 91	Linux 0.01:	bash, Dateisystem
Jan 92	Linux 0.12:	Virtueller Speicher (Paging)
Mär 92	Linux 0.95:	X-Windows, Unix Domain Sockets (jetzt fehlte nur noch Netzwerk!)
Mär 94	Linux 1.00:	**Netzwerk und TCP/IP**

Betriebssysteme ↦ ausgeforscht?

„Systems Software Research is Irrelevant" [9]

Urgestein Robert Pike (2000), einer der Entwickler von UNIX, Inferno [5], Plan 9 [10] und UTF-8 (zur Zeit bei Google beschäftigt):

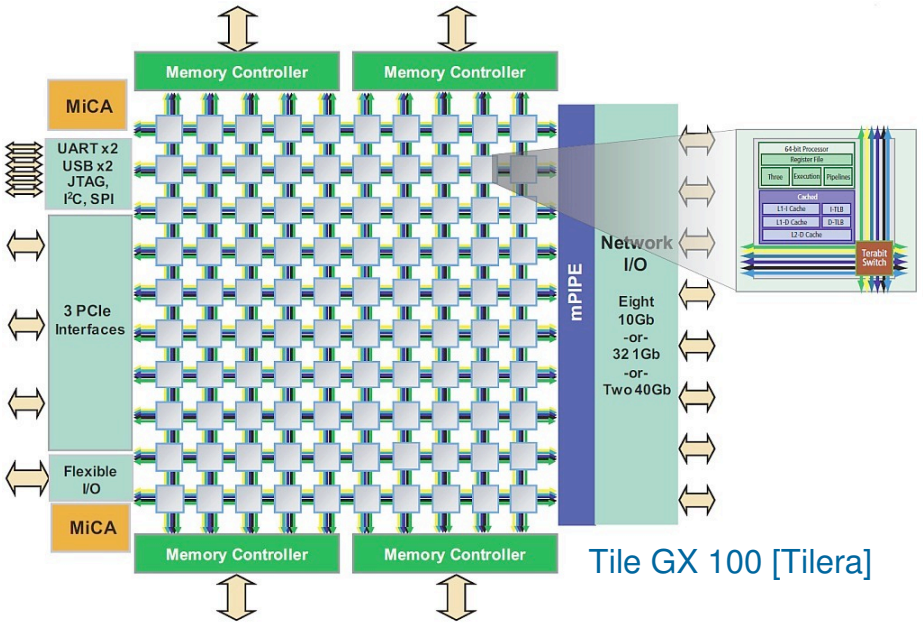
- Where is the innovation? ⇝ Microsoft, mostly
- Every other „new" OS ends up being UNIX
- Linux? ⇝ Just another copy of the same old stuff
- . . .



New Operating Systems at SOSP [9]

Aber dann. . .

The Multicore Challenge!

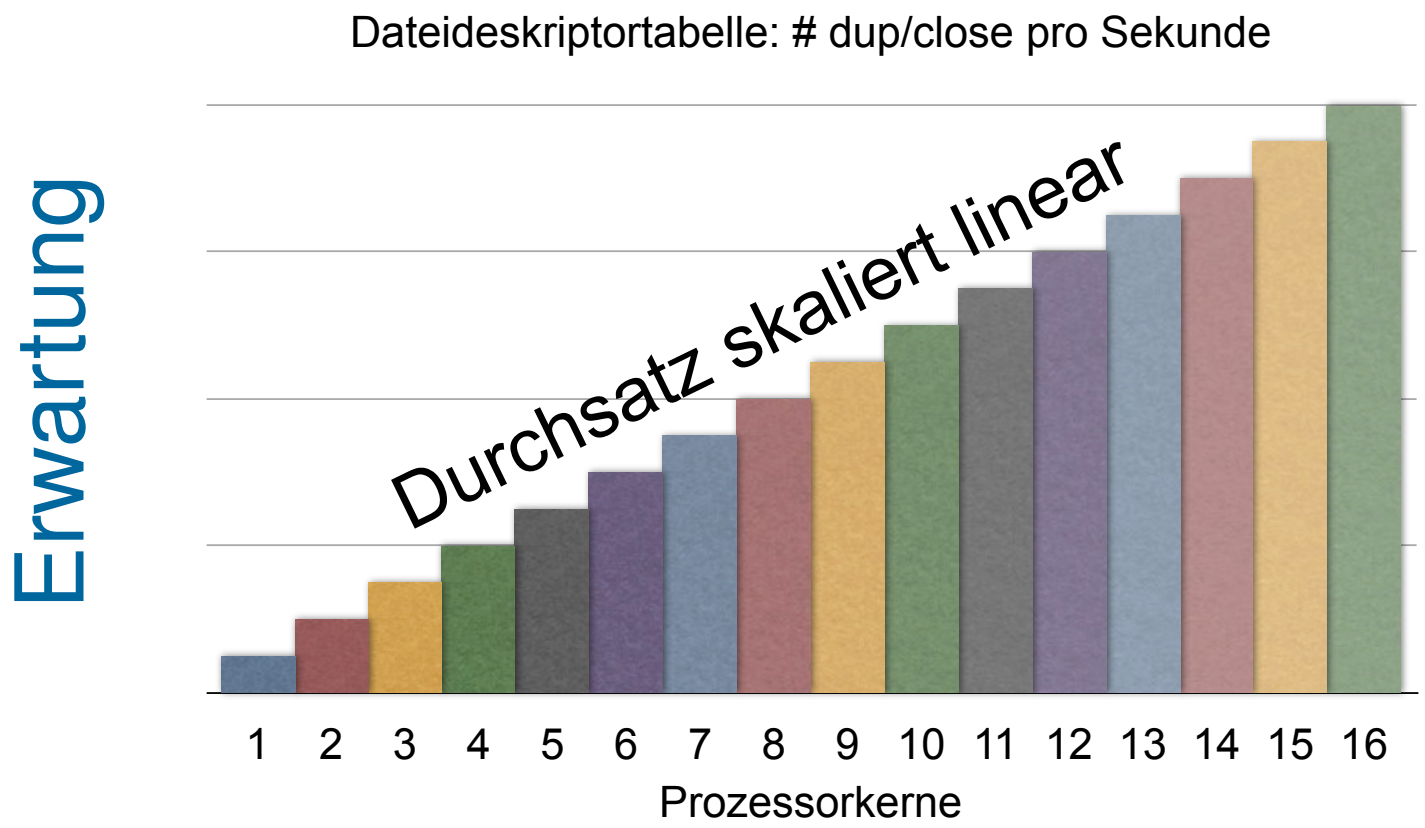


Tile GX 100 [Tilera]

Fallstudie: Dateideskriptortabelle in Linux

- Boyd-Wickizer u. a. (OSDI 2008) [2]
  - Linux 2.6.25 auf 16-Kern AMD Opteron, 1–16 Kerne in Gebrauch
  - Pro Kern ein Faden, der Dateideskriptoren anfordert und freigibt:

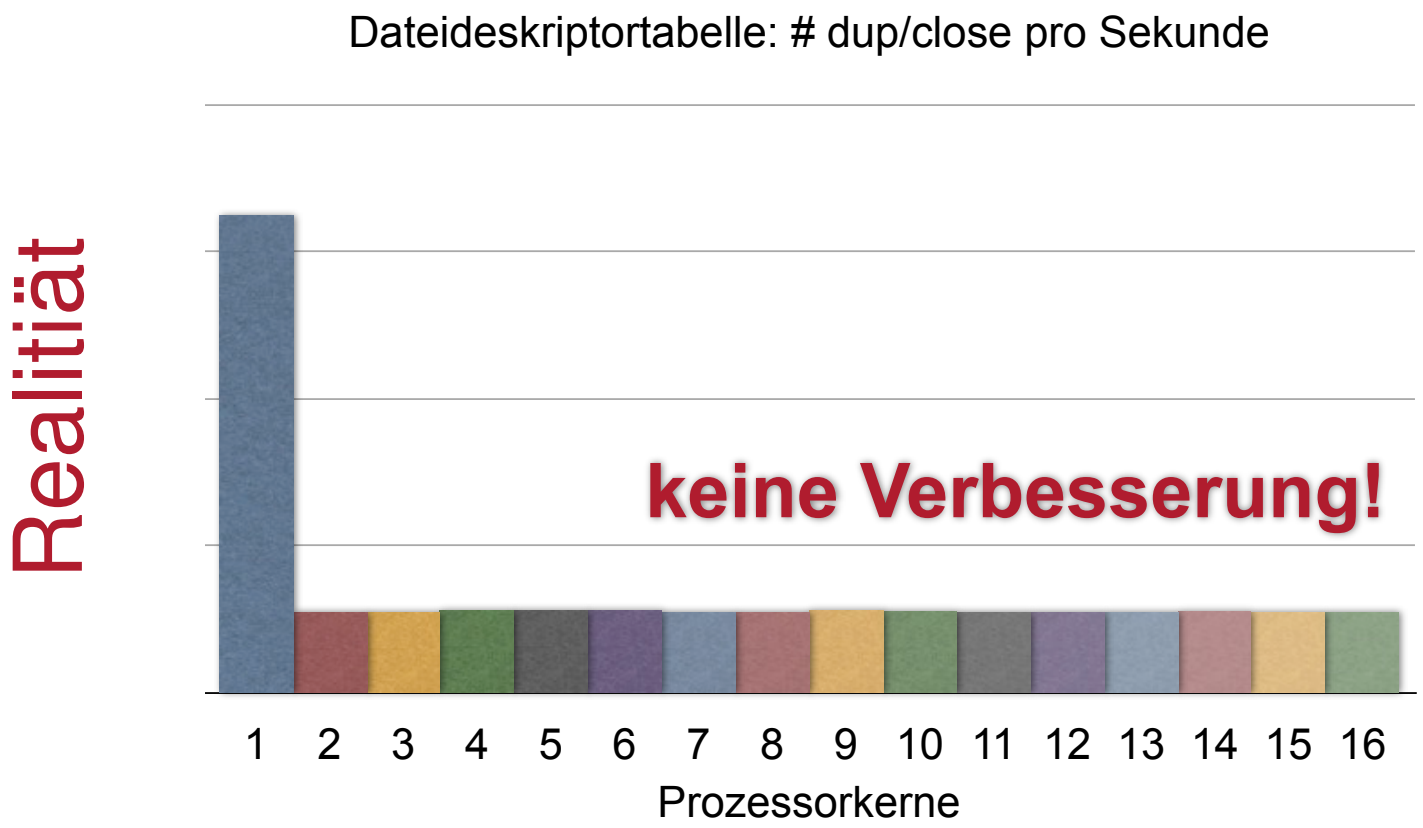
```
int f = open(...); while(1){ close( dup( f ) ); }
```



Fallstudie: Dateideskriptortabelle in Linux

- Boyd-Wickizer u. a. (OSDI 2008) [2]
  - Linux 2.6.25 auf 16-Kern AMD Opteron, 1–16 Kerne in Gebrauch
  - Pro Kern ein Faden, der Dateideskriptoren anfordert und freigibt:

```
int f = open(...); while(1){ close( dup( f ) ); }
```

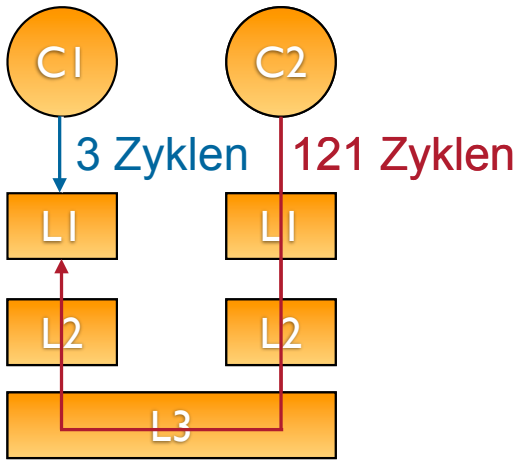


Fallstudie: Dateideskriptortabelle in Linux

- Boyd-Wickizer u. a. (OSDI 2008) [2]
  - Linux 2.6.25 auf 16-Kern AMD Opteron, 1–16 Kerne in Gebrauch
  - Pro Kern ein Faden, der Dateideskriptoren anfordert und freigibt:
    `int f = open(...); while(1){ close( dup( f ) ); }`
- Ergebnis: Schon ab **2 Kernen sinkt** der Gesamtdurchsatz
  1. Grobgranulares *Locking* ⇝ *false sharing* ⇝ keine Skalierbarkeit
  2. Geteilte Datenstruktur ⇝ *cache trashing* ⇝ Durchsatzabfall

```
fd_alloc () {
  lock(fd_table);
  fd = get_free_fd();
  set_fd_used(fd);
  fix_smallest_fd();
  unlock(fd_table);
}
```

1. false sharing



2. cache trashing

Fallstudie: Dateideskriptortabelle in Linux

- Boyd-Wickizer u. a. (OSDI 2008) [2]
  - Linux 2.6.25 auf 16-Kern AMD Opteron, 1–16 Kerne in Gebrauch
  - Pro Kern ein Faden, der Dateideskriptoren anfordert und freigibt:
    ```
    int f = open(...); while(1){ close( dup( f ) ); }
    ```
- Ergebnis: Schon ab **2 Kernen sinkt** der Gesamtdurchsatz
  1. Grobgranulares *Locking* $\rightsquigarrow$ *false sharing* $\rightsquigarrow$ keine Skalierbarkeit
  2. Geteilte Datenstruktur $\rightsquigarrow$ *cache trashing* $\rightsquigarrow$ Durchsatzabfall

Multicore: POSIX ($\mapsto$ UNIX) considered harmful!

*„This problem is not specific to Linux, but is **due to POSIX semantics**, which require that a new file descriptor be visible to all of a process's threads even if only one thread uses it."* [2]

Folgerung: Wir brauchen neue Entwurfsansätze!

- Corey — MIT, OSDI 2008, Exokern-artig: [2]
  - *Sharing* unter die Kontrolle der Applikation stellen
  - Datenstrukturen (im Normalfall) nur von einem Kern aus bearbeiten
  - Anwendungen müssen angepasst werden
- Barrelfish — ETH/MSR, SOSP 2009, Mikrokern-artig: [1]
  - BS als verteiltes System von Kernen verstehen und organisieren
  - kein implizites *Sharing*, Kommunikation nur über Nachrichten
- Factored OS (fos) — MIT, 2009, Mikrokern-artig: [15]
  - BS für 100 bis 1000 Kerne $\leadsto$ *time sharing* wird zu *space sharing*
  - Letztlich ähnlicher Ansatz wie Barrelfish
- TxOS — UT, SOSP 2009, Monolith (Linux): [11]
  - Konkurrenz zulassen durch *transactional syscalls* (statt *Locks*)
  - Anwendungen müssen angepasst werden

. . . oder doch nicht?

- Boyd-Wickizer u. a. (OSDI 2010) [3]
  - „An Analysis of Linux Scalability to Many Cores“
  - Skalierbarkeit von Linux 2.6.35-rc5 auf 48-Kern AMD Opteron
- Ansatz: *run* – *analyze* – *fix*
  - *run:* sieben „systemintensive” Anwendungen
    - Exim, memcached, Apache, PostgreSQL, gmake, Psearchy, MapReduce
  - *analyze:* gezielte Identifizierung von Flaschenhälsen
    - im Linux-Kern selber (16)
    - im Entwurf der Anwendung
    - durch die ungeschickte Verwendung der Systemschnittstelle
  - *fix:* Verbesserung, überwiegend durch Standardtechniken der parallelen Programmierung ($\rightsquigarrow$ [PFP])

. . . oder doch nicht?

- Clements u. a. (SOSP 2013) [4]
  - „The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors“
  - Skalierbarkeit von *Schnittstellen* theoretisch und praktisch untersucht anhand Kommutativität der (möglichen) Implementierung.

- Idee: Wenn Operationen kommutativ sind, können sie (im Prinzip) auch skalierbar implementiert werden.

. . . oder doch nicht?

Ergebnis: Alles nicht so schlimm. . .

*„We find that we can remove most kernel bottlenecks that the applications stress by modifying the applications or kernel slightly. [. . .] the results suggest that **traditional kernel designs may be compatible with achieving scalability** on multicore computers.”* [3]

*„Finally, using sv6, we showed that it **is practical to achieve a broadly scalable implementation of POSIX** by applying the rule, and that commutativity is essential to achieving scalability and performance on real hardware. ”* [4]

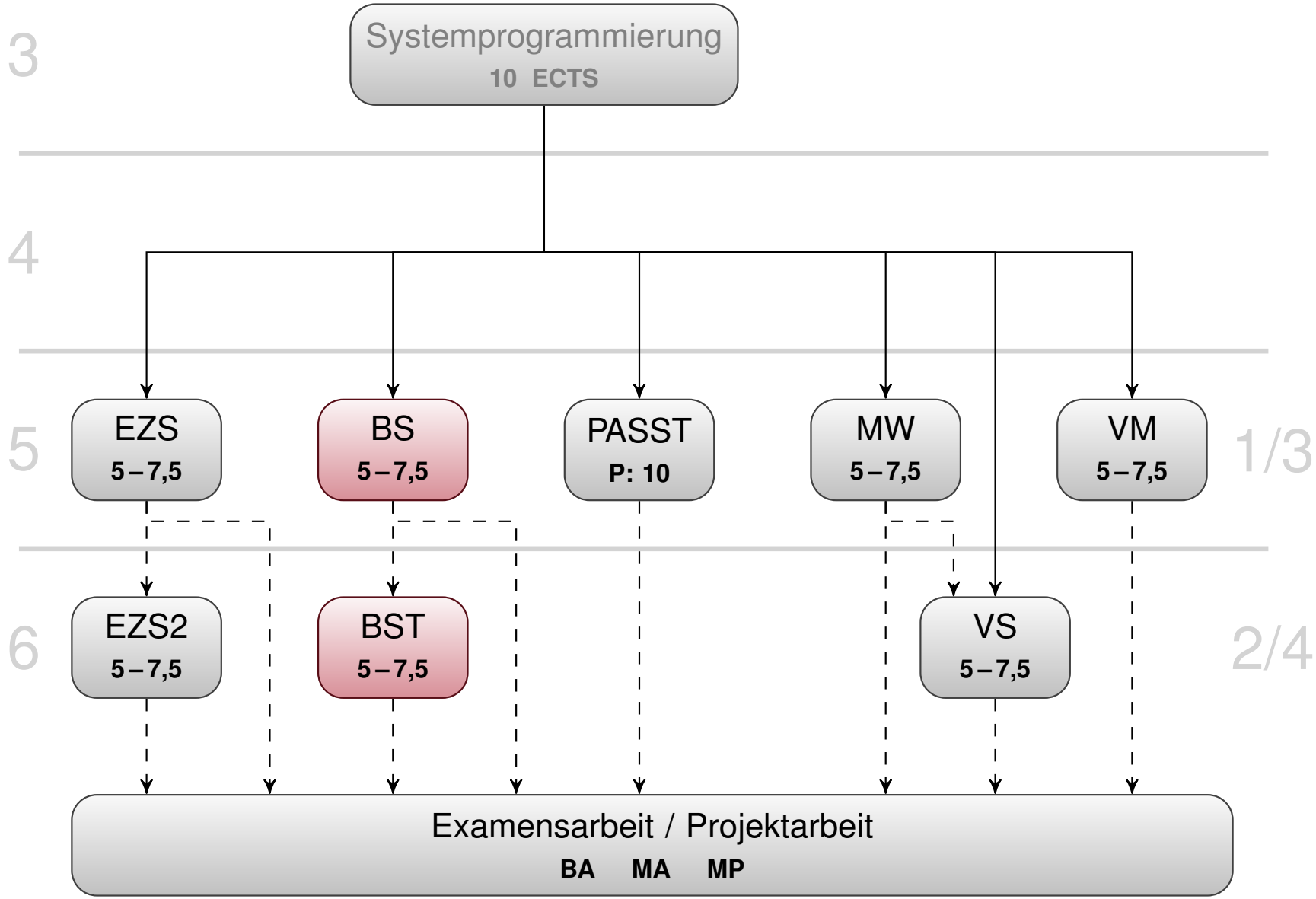
Fazit

Es bleibt spannend!

Systementwurf für Skalierbarkeit ⇝ [CS] (WS 2021).

Wie geht es weiter?

(Bachelor/Master)



Lernziele

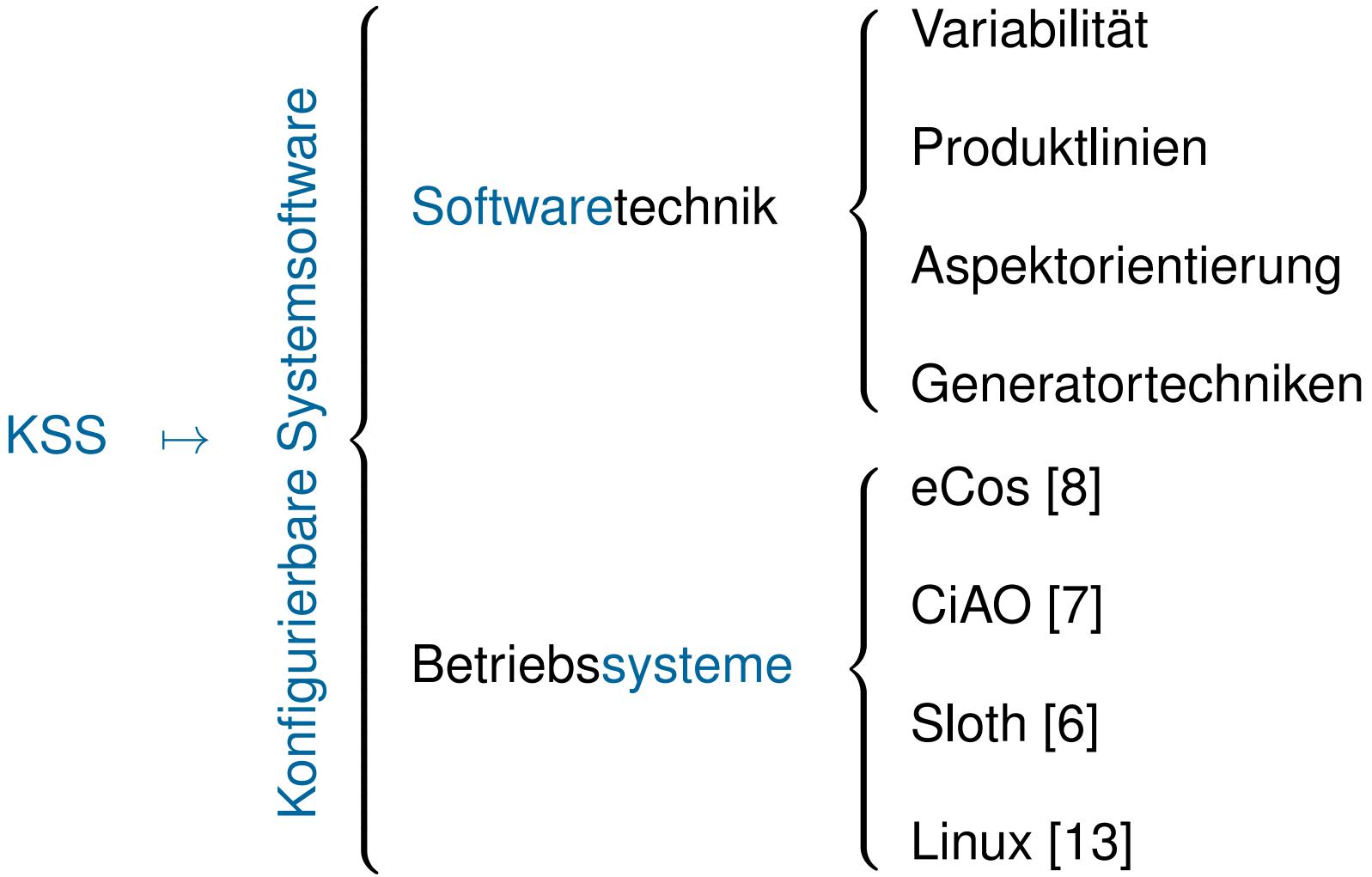
Vorlesung

- *Wissen* zu Adressraumkonzepten von Betriebssystemen vertiefen
- *Verstehen* über (logische) Adressräume festigen
  - inhaltliches Begreifen verschiedener Facetten von Adressräumen
  - intellektuelle Erfassung des Zusammenhangs, in dem Adressräume stehen

Übung $\rightsquigarrow$ mikrokern-ähnliches Betriebssystem

- *Anwenden* ausgewählter Vorlesungsinhalte für OOStuBS
- *Analyse* der Anforderungen an und Gegebenheiten von OOStuBS
- *Synthese* von Adressraumabstraktionen und OOStuBS
- *Evaluation* des erweiterten OOStuBS: Vorher-nachher-Vergleich

Hinter der Kulisse: KSS in aller Kürze...



Motivation: Special-Purpose Systems

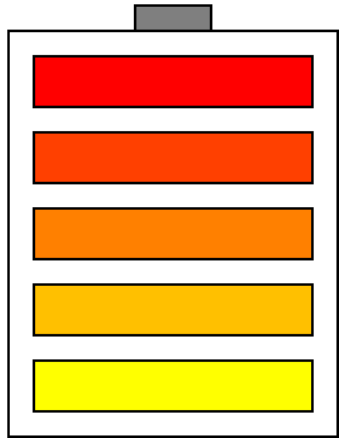
SLOTH: Threads as Interrupts

- **Idea: threads are interrupt handlers, synchronous thread activation is IRQ**
- Let interrupt subsystem do the scheduling and dispatching work
- Applicable to priority-based real-time systems
- Advantage: small, fast kernel with unified control-flow abstraction

Ausblick: Power-Aware Critical Sections

Spin-Locks brauchen ggf. viel Energie

- Sleeping-Locks besser?
- Lock-free Algorithmen besser?
- Wait-free Algorithmen besser?
- ...

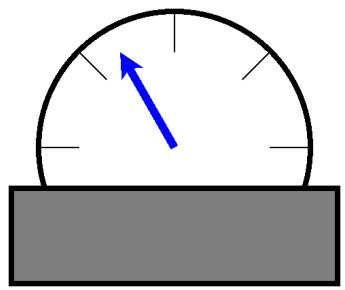


Ausblick: Power-Aware Critical Sections

Problem: wie misst man den Energieverbrauch *einer* Applikation?

Zuweisung schwierig

- P_{cpu} <=> App
  Scheduling, kritische Abschnitte, ...?
- P_{mem} <=> App
  Kernel-Verwaltungsstrukturen, Buffer-Cache, ...?
- $P_{I/O}$ <=> App
  Nebenläufigkeit, Interrupts, ...?



Ausblick: Power-Caping

Stromerzeuger wollen gleichmäßigen Verbrauch

- hoher Strompreis, wenn viele Verbraucher
- niedriger (z.T. negativer!) Strompreis, wenn wenige Verbraucher

Daher:
=> Rechenzentren zum Ausgleichen
=> Rechen-Aufträge rund um den Globus verschicken

- „Strafe” bei stark schwankendem Verbrauch

Daher:
=> BS soll Stromverbrauch „kappen”/regeln

Quelle: Wikipedia

Ausblick: Frequenz <=> Power

Energieverbrauch:

$$E = \int_t P_t dt$$

Leistungsaufnahme:

$$P_t \propto V_t^2 f_t$$

Spannung V muss bei höherer Frequenz f höher sein.

Daher: statt einer CPU besser 2 CPUs mit halber Frequenz betreiben

Ausblick: JITTY

Linux:
handoptimiert (Code, Cache)

- \- riesig
- \- unübersichtlich
- \- unwartbar
- ? (un)sicher
- \+ schnell

„schönes” OS:
„sauberer” Code

- \+ klein
- \+ lesbar
- \+ wartbar
- \+ sicher
- \- langsam

JITTY: Just-In-Time-compiliertes BS

Wie alles anfing...

Gesucht: Code-Beispiele aus bekannten Betriebssystemen für Betriebssystem-Vorlesung

Problem:

Linux:	Makro-/`#ifdef`-Hölle, selbst-modifizierender Code
*BSD:	viele Makros / viel `#ifdef`
Minix:	Mikro-Kern
...:	...
Windows:	keine Sourcen einsehbar

Examensarbeiten am LS4

Zur Zeit im Angebot:

- Bachelorarbeiten
- Masterarbeiten

`https://sys.cs.fau.de/theses`

... oder persönlich nachfragen...!

Das war's :-)

Das LS 4 BS-Team wünscht
erfolgreiche und erholsame
„Semesterferien“

. . . und ein Wiedersehen
im Sommersemester 2024!

Referenzen

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand u. a. „The multikernel: a new OS architecture for scalable multicore systems“. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. ACM Press. Big Sky, MT, USA: ACM Press, Okt. 2009, S. 29–44. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen u. a. „Corey: An Operating System for Many Cores“. In: *8th Symposium on Operating System Design and Implementation (OSDI '08)*. USENIX Association. San Diego, CA, USA: USENIX Association, Dez. 2008, S. 43–57. ISBN: 978-1-931971-65-2. URL: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/boyd-wickizer/boyd_wickizer.pdf.

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao u. a. „An Analysis of Linux Scalability to Many Cores“. In: *9th Symposium on Operating System Design and Implementation (OSDI '10)*. USENIX Association. Vancouver, BC, Canada: USENIX Association, Okt. 2010. ISBN: 978-1-931971-79-9.

Referenzen (Forts.)

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich u. a. „The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors“. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, USA). ACM Press. New York, NY, USA: ACM Press, 2013, S. 1–17. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522712.

Sean Dorward, Rob Pike, Dave Presotto u. a. „The Inferno Operating System“. In: *Bell Labs Technical Journal* 2.1 (1997). URL: http://www.vitanuova.com/inferno/papers/bltj.html.

Wanja Hofer, Daniel Lohmann, Fabian Scheler u. a. „Sloth: Threads as Interrupts“. In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)* (Washington, D.C., USA, 1. Dez. 2009–4. Dez. 2009). IEEE Computer Society Press, Dez. 2009, S. 204–213. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.

Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat u. a. „CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems“. In: *Proceedings of the 2009 USENIX Annual Technical Conference*. San Diego, CA, USA: USENIX Association, Juni 2009, S. 215–228. ISBN: 978-1-931971-68-3. URL: https://www.usenix.org/legacy/event/usenix09/tech/full_papers/lohmann/lohmann.pdf.

Referenzen (Forts.)

Daniel Lohmann, Fabian Scheler, Reinhard Tartler u. a. „A Quantitative Analysis of Aspects in the eCos Kernel“. In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. Hrsg. von Yolande Berbers und Willy Zwaenepoel. Leuven, Belgium: ACM Press, Apr. 2006, S. 191–204. ISBN: 1-59593-322-0. DOI: 10.1145/1218063.1217954.

Norbert Oster. *Parallele und Funktionale Programmierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 2, 2015 (jährlich). URL: https://www2.cs.fau.de/teaching/SS2015/PFP/index.html.

Rob Pike. *Systems Software Research is Irrelevant*. Talk. CS Colloquium, Columbia University. URL: http://herpolhode.com/rob/utah2000.pdf (besucht am 09. 12. 2010).

Rob Pike, Dave Presotto, Sean Dorward u. a. „Plan 9 from Bell Labs“. In: *Computing Systems* 8.3 (1995), S. 221–254.

Referenzen (Forts.)

Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach u. a. „Operating System Transactions“. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. ACM Press. Big Sky, MT, USA: ACM Press, Okt. 2009, S. 161–176. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629591.

Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09). ACM Press. Big Sky, MT, USA: ACM Press, Okt. 2009. ISBN: 978-1-60558-752-3.

Wolfgang Schröder-Preikschat. *Betriebssystemtechnik*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/SS15/V_BST.

Wolfgang Schröder-Preikschat. *Concurrent Systems*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_CS.

Referenzen (Forts.)

Reinhard Tartler, Daniel Lohmann, Julio Sincero u. a. „Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem“. In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)* (Salzburg, Austria). Hrsg. von Christoph M. Kirsch und Gernot Heiser. New York, NY, USA: ACM Press, Apr. 2011, S. 47–60. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966451.

Linus Torvalds und David Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperCollins, 2001. ISBN: 978-0066620725.

David Wentzlaff und Anant Agarwal. „Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores“. In: *ACM SIGOPS Operating Systems Review* 43 (2 Apr. 2009), S. 76–85. ISSN: 0163-5980. DOI: 10.1145/1531793.1531805.